A Specification Invariant Technique for Operation Cost Minimisation in Flow-graphs *

Martin Janssen[1] Francky Catthoor[1,2] Hugo De Man[1,2]

[1]IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
[2]Professor at the Katholieke Universiteit Leuven

Abstract

In high-level synthesis, optimising area, time, and power in real-time applications are the prime objectives. In this paper, a new model and technique are proposed, which minimise a weighted operation cost function for data-paths at an early stage in the synthesis process. Our main target domain consists of lowly-multiplexed and hard-wired implementations of real-time DSP applications. The behavioral specification of an application is translated into a signal flow-graph, the operation cost (area/power) of which is minimised using algebraic transformations. A minimal set of elementary transformations are combined in a flexible way into composite transformations, which are used in an ascent/steepest descent search algorithm. Experiments show that this approach achieves optimal or close to optimal results with very few transformations. The technique is invariant to changes in the specification, as long as these retain the bit-true input/output behaviour.

1 Introduction and related work

A system designer crafting a DSP subsystem faces a large design space at the specification level. This search space is hierarchically structured. At the top levels, the options are related to significant distinctions ,such as the difference between DFT and FFT for Fourier transform. In contrast, the lower levels in the design space typically reflect input/output behaviour preserving and local structural changes (like rewriting $a \cdot b + a \cdot c$ to $a \cdot (b + c)$), which optimise the mapping from the specification to the final implementation. We believe however that exploring these lower level alternatives can be done also automatically with a data-path cost optimisation tool that is invariant to functionally equivalent structural changes in the specification. With such an approach, a system designer can concentrate on more important higher-level trade-offs.

Transformations are often used for optimisation purposes. In parallel compilers, transformations are used to exploit parallelism in flow-graphs [1]. In high-level synthesis, transformations are mainly used to optimise throughput [2, 3]. Recently, transformations are also used in power optimisation [4], where the steering is limited to a generic global optimisation technique on a subset of the possible transformations. Also using transformations for direct area optimisation has not yet attracted much attention. Local resource utilisation optimisation has been done using transformations steered by stochastic techniques [5]. Sometimes area optimisation is a secondary goal in throughput optimisation [5].

Because most transformations are well-known [6, 7], selecting a set of transformations is relatively simple. It is however extremely important to reduce the set to a minimal size to limit the search space. This requires an appropriate model. A transformation can be executed when all its preconditions are satisfied. However, doing so may enable or disable other transformations. This dependency makes finding an efficient and effective ordering both desirable and very difficult in almost all cases. Therefore, most research relies on manually selected orderings [7, 8]. One way to tackle the ordering problem is by executing random moves using simulated annealing [5]. However, a stochastic method such as simulated annealing is based on the assumption that many small and independent moves can be performed very fast. But transformations are dependent, and their applicability always has to be checked prior to execution, which makes them relatively slow. The ordering problem can be tackled by formalising the preconditions and postconditions of a limited set of transformations [9]. From this, the enabling and disabling relationships between transformations are derived, and represented in a graph. An ordering is obtained from levelling the graph. Cycles in the graph are resolved by allowing the 'most important' transformation to come first. Unfortunately, the transformations addressed in this approach are mainly loop transformations.

Area minimisation is a well-known goal also in multilevel logic optimisation [11], which operates at the bit-level. We work at the word-level though, with many (user-defined) operation types. The transformations used here should therefore be defined in terms of (a large set of) operation properties and not only operation types like AND/OR/NOT. Multi-level logic optimisation could be applied to an application by expanding the word-level description application to the bit-level. But then other word-level tasks, such as sharing hardware, cannot be performed efficiently anymore [12]. Moreover, fast system level ex-

*This research was sponsored by the ESPRIT 2260 ("SPRITE") project of the EC.

ploration would be unfeasible due to the complexity (also due to the presence of e.g. variable multiplications).

The goal in this paper is weighted operation cost minimisation for data-paths in lowly-multiplexed and hard-wired implementations of real-time DSP applications. Doing this at an early stage in the design process will have the largest impact. The problem is formulated as the minimisation of a weighted sum of all operation costs in a signal flow-graph (SFG) by means of algebraic transformations. The transformation ordering problem is tackled as follows in a new approach. In a first step, called *normalisation*, the initial SFG is transformed into an SFG of maximum cost using an ascent algorithm. This is based on a, for our target class, novel model which exhibits the important property that the maximum cost SFG is the same (invariant) for all SFG's with the same input/output behaviour. In a second step, called *optimisation*, the maximum cost SFG is transformed into a SFG of minimum cost using a steepest descent algorithm. This approach can only work with transformations which are powerful enough to 'lookahead' over small bumps in the down-hill path, i.e. which are able to execute other (zero- or negative-gain) transformations such that the transformation itself becomes valid and yields a positive gain. The limited lookahead transformations are a subset of the elementary transformations. Their preconditions are weaker. Instead of demanding that all preconditions are satisfied, an analysis is made to see if unsatisfied preconditions can be 'repaired' by first executing other transformations. If not, the transformation cannot be executed. If all unsatisfied preconditions can be repaired, the transformation can be executed after the other transformations have been executed. Limited lookahead transformations are composed out of other transformations, therefore they will also be referred to as *composite* transformations. They enable the use of a greedy search strategy with very few moves.

Before explaining our approach in detail in section 4, first the class of signal flow-graphs used in this approach are introduced in section 2, followed by a discussion on the set of currently supported transformations in section 3.

2 Target signal flow-graphs

2.1 Definitions

In our context, a signal flow-graph G is a tuple (V, E) where V is the set of nodes and E is the set of signals. A signal $e \in E$ is a directed hyper edge which has exactly one source node and zero or more sink nodes. Signals carry a data value from the source node to all the sink nodes. There are four node types: constant, input, operation, and output.

An operation $\otimes : S \times S \rightarrow S$ is called a *binary operation* on domain S. More generally, for any positive integer n, an operation $\otimes : S^n \rightarrow S$, is called an *n-ary operation* on S. An operation $\odot : S \times T \rightarrow S$ with $(x, y) \mapsto x$ for all $(x, y) \in S \times T$, is called a *projection* from $S \times T$ onto S.

For two binary operations $\otimes : S \times S$ and $\oplus : S \times S$ on domain S, $\otimes$ is *left-distributive* over $\oplus$ iff $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ for all $x, y, z \in S$; *right-distributive* over $\oplus$ iff $(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$ for all $x, y, z \in S$. If $\otimes$ is both left- and right-distributive over $\oplus$, then $\otimes$ is said to be *distributive over* $\oplus$.

For an SFG with n primary inputs in domain S and m primary outputs in domain S, the bit-true input/output behaviour is a function $f : S^n \rightarrow S^m$. Transformations on an SFG are *input/output behaviour preserving*, if they do not change f.

2.2 Operations

Initially, an SFG can contain the following arithmetic operations: addition, negation, subtraction, constant and variable multiplication, upshift, downshift, and constant exponentiation. In addition, the delay operation is supported. However, cycles are not allowed in an SFG. Typically, the above operations have one or two operands, but this is not a restriction. To reduce the number and complexity of the algebraic transformations, the above operations are modelled using only three *basic* operations: *addition* $(+)$, *multiplication* $(\times)$, and *delay* $(\triangle)$. The delay operation is not only used to refer to a previous value of a signal in *time* (known as *sample delay* or *algorithmic delay*), but also to the value of a signal in a previous iteration of a loop which is not the time loop.

Addition and multiplication are n-ary operations ($n \geq 2$). Delay is a projection from $S \times \mathbb{N}$ to S. Addition and multiplication are both commutative and associative, delay is neither commutative nor associative. Multiplication is distributive over addition, and delay is right distributive over multiplication and addition.

In the initial SFG, operations which are not basic will be substituted by a combination of basic operations. An example is the substitution of $a - b$ by $a + (b \times (-1))$. Substitution is an input/output behaviour preserving transformation. In addition, substitution is cost preserving, e.g. in the example the cost of the addition and multiplication is equal to the cost of the subtraction. (See section 2.4.)

2.3 Signal types and type propagation

In signal processing applications, invariant word-lengths are frequently used, e.g. the word-length of the output signal of an addition or multiplication is equal to the word-length of the input signal(s). This may cause overflow. Applying algebraic transformations to signal flow-graphs with potential overflow means that preserving the input/output behaviour of the signal flow-graph cannot be guaranteed, and simulation is needed to check if the result is still acceptable.

To be able to guarantee input/output behaviour preservation under algebraic transformations, full precision arithmetic is needed. Full precision arithmetic requires a type propagation mechanism which propagates signal types (word-lengths) from input(s) to output(s) of a signal flow-graph. A 'classic' way of type propagation is the *logarithmic method*, in which the word-lengths of the input signals of an operation are used to compute the word-length of the output signal. For example, an addition with two 8 bits wide input signals will have a 9 bits wide output signal. The logarithmic method is a conservative method which prevents overflow everywhere. However, it is too conservative in that it overestimates the word-lengths of signals. In fig. 1(a), this is illustrated with an adder tree which is transformed into an adder chain using associativity. With the logarithmic method the output word-length of the chain is 11 bits, while the output word-length of the tree is 10 bits.

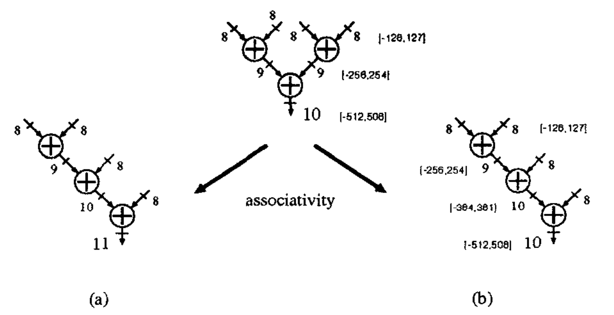

Figure 1. Type propagation with (a) the logarithmic method, and (b) the linear method.

To solve this problem, a more accurate type propagation method is used, referred to as the *linear method*. In this method, signal types are extended with a *range*. A signal range is specified by an upper- and lower-bound value for that signal. The ranges of the input signals of an operation are used to compute the range of the output signal, based on the behaviour of the operation. The word-length of a signal can be derived from its range. This is illustrated in fig. 1(b). With the linear method the output word-length of the chain is equal to the output word-length of the tree. Moreover, it is minimal when overflow is to be excluded.

2.4 Cost function

The cost of an SFG is an estimate of the area of the cheapest hard-wired implementation in hardware, and is defined as the sum of the cost of each node in the SFG. The cost of a node is the cost of its cheapest implementation in hardware, and depends on its surroundings. For example, the cost of a constant multiplication is the cost of the add/sub/shift network which can perform this multiplication, whereas the cost of a variable multiplication is the cost of a multiplier. The cost of a 2-input multiplication with both inputs constant is zero. With this cost function there is no need to perform operation expansion or redundancy removal prior to cost calculation.

Besides area, also power can be expressed in the cost function in much the same way as area is expressed. The proposed minimisation technique will remain the same.

3 Transformations

3.1 Elementary transformations

A minimal set of elementary transformations is defined for the area minimisation problem. An important aspect of this set is that most transformations have a *reverse*. An example is *common subexpression replication*, which is the reverse of *common subexpression elimination* [5]. The set of transformations considered in our prototype implementation is listed below.

eliminate and **replicate.** *Eliminate* is common subexpression elimination. An example is $c = a + b, d = a + b \Longrightarrow c = a + b, d = c$. *Replicate* is the reverse of *eliminate*.

collectR and **distributeR.** *CollectR* is a transformation based on right distributivity. An example is $(a \times c) + (b \times c) \Longrightarrow (a + b) \times c$. *DistributeR* is the reverse of *collectR*.

commute. *Commute* is a transformation based on commutativity. An example is $a + b \Longrightarrow b + a$. *Commute* is its own reverse.

mergeC and **splitC.** *MergeC* is a transformation that merges operations of the same type which are commutative and associative. An example is $(a + b) + c \Longrightarrow a + b + c$. This transformation removes any ordering of operations of the same type. *SplitC* is the reverse of *mergeC*. This transformation imposes a (partial) ordering on operations of the same type.

mergeNC and **splitNC.** *MergeNC* is a transformation that merges a chain of operations of the same type, which are neither commutative nor associative, into one operation. An example is $(a \bigtriangleup 1) \bigtriangleup 2 \Longrightarrow a \bigtriangleup (1 + 2)$. *SplitNC* is the reverse of *mergeNC*.

compute and **computeReverse.** *Compute* is a transformation which performs constant computation (also known as constant folding). An example is $1 + 2 \Longrightarrow 3$. *ComputeReverse* is the reverse of *compute*. This transformation is useful for performing additive or multiplicative decompositions on constants.

pass and **passReverse.** *Pass* is a transformation based on the neutral element of an operation. An example is $a \times 1 \Longrightarrow a$. *PassReverse* is the reverse of *pass*.

block. *Block* is a transformation based on the zero element of an operation. An example is $a \times 0 \Longrightarrow 0$.

eliminateDeadCode. This transformation removes any node whose signal is not used somewhere else in the SFG.

The last two transformations are only intended for redundancy cleanup and do not have a reverse.

The elementary transformations are defined in terms of operation properties, such as right-distributivity, and operate on the basic operations. They are, however, not restricted to these operations. Since the delay operation is right-distributive over addition and multiplication, the *collectR* and *distributeR* transformations can be applied to it.

For example, $(a \triangle 1) + (b \triangle 1) \Longrightarrow (a+b) \triangle 1$. This is in fact a retiming transformation [10], but used in a different context (area minimisation vs. critical path optimisation).

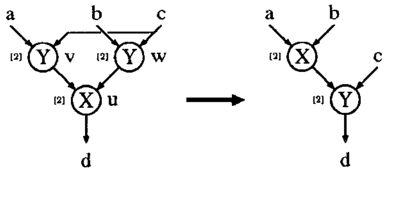

Figure 2. The elementary collectR transformation.

Elementary transformations consist of three parts: (i) *preconditions* that have to be satisfied for the transformation to be valid on a certain subgraph, (ii) a *subgraph creation/substitution mechanism*, and (iii) a *cost*. The cost of a transformation is defined as the difference in cost between its target and source subgraphs. Preconditions describe a kind of pattern matching on a subgraph, which is very strict for the elementary transformations. For example, signal *c* in fig. 2 is not allowed to be located at the first input of operations *v* or *w*. These strict preconditions make the elementary transformations unsuitable for direct application to an SFG. However, more powerful transformations can be composed from the set of elementary transformations in a very flexible way. This in contrast to having a large library with a (virtually unlimited) variety of highly specialised transformations.

3.2 Composite transformations

Elementary transformations perform the actual SFG transformations. The effect of each individual transformation is known. For the minimisation problem it is also known which transformations have to be performed. However, due to the strict preconditions of the elementary transformations, their applicability is very low. Other 'enabling' transformations have to be executed to improve the applicability, but which transformations and in what order? This problem is solved using *composite transformations* which perform a 'limited lookahead' to be able to execute 'enabling' transformations only when they are needed.

Composite transformations are a subset of the elementary transformations. Based on the knowledge of what the other available transformations can do, their preconditions have been relaxed, such that not all have to be satisfied immediately. For each unsatisfied precondition that is allowed to occur, a *fixed* (sequence of) transformation(s) is defined, which enables the unsatisfied precondition. The sequence of transformations that results from *several* unsatisfied preconditions is not fixed, i.e. it is dynamically composed out of the individual precondition enabling transformation sequences. This makes the composite transformations both powerful and flexible. In addition to 'enabling' transformations, also 'clean-up' transformations are included in the above sequences. The cost of a composite transformation is defined as the sum of the cost of each elementary transformation in a full sequence.

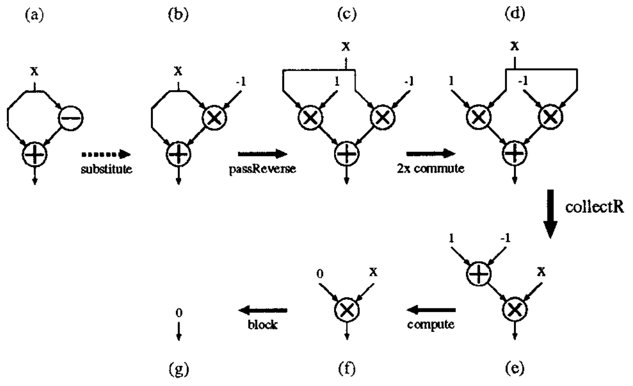

Figure 3. An execution sequence of the composite *collectR* transformation.

EXAMPLE 1. Consider expression $x + (-x)$, which is depicted in fig. 3(a). This SFG contains a negate operation, which is substituted in fig. 3(b). When the composite *collectR* transformation is evaluated for the addition in this SFG, then a number of preconditions are satisfied, such as a right-distributive operation (multiplication) at one of its inputs, and the existence of a common signal x. However, other preconditions are not satisfied: there must be a multiplication at both inputs of the addition, and the common signal must be located at the second input of the multiplications. This situation can be repaired though, by first performing a *passReverse* transformation to introduce the second multiplication (fig. 3(c)), followed by two *commute* transformations (fig. 3(d)). Now all preconditions are satisfied and the elementary *collectR* transformation can be executed (fig. 3(e)). Finally, a *compute* transformation (fig. 3(f)) and a *block* transformation (fig. 3(g)) are executed to clean up the SFG. In this example, six elementary transformations are needed for one composite *collectR* transformation.

4 The minimisation technique

The minimisation technique consists of four steps: (i) substitution, (ii) *normalisation*, (iii) *optimisation*, and (iv) back substitution. Substitution is explained in section 2.2. Back substitution is its reverse.

4.1 Normalisation

Normalisation serves two purposes: (i) to have a *specification invariant* starting point for optimisation, and (ii) to have a starting point that can lead to a good (if not the best) solution with a limited search during optimisation. This is achieved by normalising to a point in the search space which is located 'above' the global optimum so that a 'down-hill' path can be found. To obtain these properties we propose to transform the initial specification to a normalised SFG, for which: (i) every operation has a fanout of one, (ii) on any path from input to output, at most 3 different operations reside, in the order delay, multiplication, addition (multi-input operations count as one), and (iii) there are no redundancies due to reconvergent input signals. A normalised SFG has maximum cost and maximum parallelism. It is not unique, e.g. input signal ordering is not performed. But this does not affect optimisation results.

In pseudo code, normalisation is expressed as follows:

```
normalise() {
    do_cleanup();
    do_merge();
    do {
        do_distributeR();
        do_replicate();
    } while( 'change' );
}
```

In *do_cleanup()*, *compute*, *pass*, and *block* remove redundancies in the SFG. In *do_merge()*, operations are merged using *mergeC* and *mergeNC*. In the main loop, the desired ordering of operations is obtained by repeatedly executing *distributeR*, and the fanout of operations is reduced to one by repeatedly executing *replicate*. For normalisation it suffices to execute *distributeR* and *replicate* in random order, i.e. whenever they are valid, and independent of the cost.

4.2 Optimisation

Starting from a maximum cost SFG, optimisation has to find the minimum cost SFG. In terms of composite transformations, which are able to take small 'hills' in the search space, we believe that there is always a direct down-hill path from a maximum cost SFG to the minimum cost SFG. Intuitively, it results from the characteristic that for any three states A, B, and C in the search space connected by a path with down-hill moves between A and B and an up-hill move between B and C, and C located down-hill from A, there is also a connecting path available between A and C with only down-hill moves. Experimental results have confirmed this conjecture. Therefore, we have to traverse only the set of all down-hill paths. Unfortunately, this search space is still too large for larger examples, so we have introduced further heuristics to reduce it to a steepest descent algorithm.

Only two composite transformations can perform a (large) down-hill move: *eliminate* and *collectR*. In pseudo code, optimisation is expressed as follows:

```
optimise() {
    do {
        do {
            eset = find_all_eliminate_trfs();
            if( not_empty( eset ) & cost( eset( 1 ) ) < 0 )
                eliminate_fire( eset( 1 ) );
        } while( 'echange' );
        cset = find_all_collectR_trfs();
        if( not_empty( cset ) & cost( cset( 1 ) ) < 0 )
            collectR_fire( cset( 1 ) );
    } while( 'change' );
}
```

Sets *eset* and *cset* are sorted by cost, i.e. the transformation with the highest gain (lowest cost) is the first element in the set. If this gain is positive, the corresponding transformation will be fired. The sets are obtained through an exhaustive search for all possible *eliminate* and *collectR* transformations, respectively.

5 Experimental results

The minimisation technique has been applied to a wide range of examples. In table 1, the initial area (IA), normalised area (NA), final area (FA), improvement, number of composite transformations executed during optimisation, and the CPU time, are presented for some examples. The examples in the first part of the table demonstrate the specification invariant property, while the examples in the second part show the optimisation capabilities of the technique. It was implemented in C++ and was run on an HP 735 workstation. The area measures used are extracted from the building blocks of a 1.2μ module library.

	IA [mm^2]	NA [mm^2]	FA [mm^2]	imp. [%]	# CTs	CPU [s]
mask	26.88 (3.02)	110.3	24.72 (0.86)	8 (72)	22	15.0
maskp	26.48	110.3	24.72	7	23	15.2
maskr	27.04	110.3	24.72	9	23	16.1
maskm	24.72	110.3	24.72	0	22	14.7
fir	8.04	10.39	4.54	44	26	393
fird	4.89	10.39	4.54	7	26	391
dct4	10.51	10.36	4.30	59	15	24.5
dct4i	13.43	13.85	5.76	57	16	4.9
dct8	70.49	69.38	25.90	63	79	2015
dct8i	95.73	98.40	36.49	62	88	475
wdf3i	5.21	40.91	5.02	4	82	2860

Table 1. Results of minimisation on various examples.

Example *mask* is an image processing application based on 2D masking. The initial SFG of *mask* is shown in fig. 4, the minimised SFG is shown in fig. 5. Constant *c3* ("010.1") in fig. 5 was generated by a multiplicative decomposition of *c1* ("0.0101") and *c2* ("0.11001"). Example *maskp* is the same as *mask*, except that some partial optimisations have been performed already. Example *maskr* is the same as *mask*, but with a redundancy added at the output ($2 \times out - out$). Example *maskm* is the minimised result of *mask*. Note that the largest part of the initial and final area figures of the mask examples are accounted for by two line-buffers (23.86 mm^2). For *mask*, in parentheses also the initial area, final area, and improvement are given when this area is not taken into account.

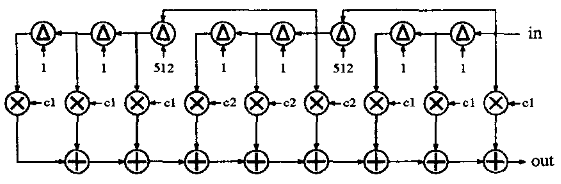

Figure 4. Initial SFG of *mask*.

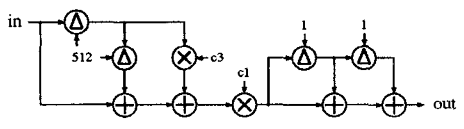

Figure 5. Minimised SFG of *mask*.

Example *fir* is a 32^{th} order symmetrical FIR filter used in an RGB-YUV codec. The largest gain is obtained by converting individual buffers originating at the input signal into a delay line. Example *fird* is the same as *fir*, except that the individual buffers have been optimised into a delay line already.

The specification invariant property of the minimisation technique is demonstrated with the normal and final area figures in the first part of table 1. For all examples they are the same. The number of composite transformations executed may differ slightly, because of the normal form which it not fully unique (see section 4.1).

Examples *dct8* and *dct4* are discrete cosine transforms. Example *dct8i* is the same as *dct8*, except that the constants are inputs. The initial SFG of *dct8i* contains 64 multiplications and 56 additions/subtractions, and the minimised SFG contains 22 multiplications and 28 additions/subtractions, which is the optimum for the given bit-true behaviour (i.e. the set of constants).

Example *wdf3i* is a 3^{rd} order wave digital filter, with the constants as inputs. It is constructed from three stages of the well-known 5^{th} order wave digital filter benchmark.

The results in the second part of table 1 show that a significant reduction in area can be obtained with very few composite transformations, and in reasonable CPU time. However, due to the exhaustive nature of the search for possible *eliminate* and *collectR* transformations, CPU times grow rapidly with an increase in number of inputs/outputs and reconverging paths.

6 Conclusions

In this paper, we have introduced an automated technique which minimises the operation cost (area/power) of arithmetic operations in SFG's by means of algebraic transformations. A minimal set of elementary transformations are combined in a flexible way into composite transformations, which are used in an ascent/steepest descent search algorithm. This approach achieves optimal or close to optimal results with very few transformations. Due to normalising the SFG prior to optimisation, the technique is invariant to functionally equivalent structural changes in the initial specification. As a result, the system designer can explore and accurately evaluate more versions of a particular algorithm without being bothered with the structural implications of the more detailed (low-level) transformations. No other methods known to us can achieve this desired property for our class of SFG's and cost function.

References

[1] C. Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on the architecture design," *IEEE Trans. on Computers*, Vol.37, No.8, pp.991-1004, Aug. 1988.

[2] A. Nicolau, R. Potasman, "Incremental Tree Height Reduction For High Level Synthesis," *Proc. of the 28th DAC*, pp. 770-774, 1991.

[3] G.P. Fettweis, L. Thiele, "Algebraic Recurrence Transformations for Massive Parallelism," *Proc. of the IEEE VLSI Signal Processing Workshop*, pp. 332-341, 1992.

[4] A. Chandrakasan, M. Potkonjak, J. Rabaey, R. Brodersen, "An Approach For Power Minimization Using Transformations," *Proc. of the IEEE VLSI Signal Processing Workshop*, pp. 41-50, 1992.

[5] M. Potkonjak, J. Rabaey, "Maximally Fast and Arbitrarily Fast Implementation of Linear Computations," *Proc. of ICCAD*, pp. 304-308, Nov. 1992.

[6] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[7] D.B. Loveman, "Program Improvement by Source-to-Source Transformation," *Journal of the ACM*, Vol. 24, No. 1, pp. 121-145, Jan. 1977.

[8] H. Samsom, L. Claesen, H. De Man, "SynGuide: An environment for doing interactive Correctness Preserving Transformations," *Proc. of the IEEE VLSI Signal Processing Workshop*, 1993.

[9] D. Whitfield, M.L. Soffa, "An Approach to Ordering Optimizing Transformations," *2nd ACM Symp. on Principles and Practice of Parallel Progr.*, pp. 137-147, Mar. 1990.

[10] C.E. Leiserson, F.M. Rose, J.B. Saxe, "Optimising Synchronous Circuitry by Retiming," *Proc. 3rd Caltech Conf. on VLSI Design*, pp. 87-116, 1983.

[11] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. on CAD*, Vol. 6, No. 6, pp. 1062–1081, 1987.

[12] W. Geurts, F. Catthoor, H. De Man, "Quadratic zero-one Programming Based Synthesis of Application Specific Data Paths", *Proc. of ICCAD*, pp. 522–525, Nov. 1993.