Maximally and Arbitrarily Fast Implementation of Linear and Feedback Linear Computations

Miodrag Potkonjak and Jan M. Rabaey

***Abstract*—By establishing a relationship between the basic properties of linear computations and eight optimizing transformations (distributivity, associativity, commutativity, inverse and zero element law, common subexpression replication and elimination, constant propagation), a computer-aided design platform is developed to optimally speed-up an arbitrary instance from this large class of computations with respect to those transformations. Furthermore, arbitrarily fast implementation of an arbitrary linear computation is obtained by adding loop unrolling to the transformations set. During this process, a novel Horner pipelining scheme is used so that the area-time (AT) product is maintained constant, regardless of achieved speed-up. We also present a generalization of the new approach so that an important subclass of nonlinear computations, named feedback linear computations, is efficiently, maximally, and arbitrarily sped-up.**

***Index Terms*—Digital signal processing (DSP), high-level synthesis, Horner scheme, linear computations, throughput, transformations, very large scale integrated circuit (VLSI) design.**

I. Introduction

THROUGHPUT is often one of the most important parameter of state-of-the art designs. Control and data dependencies impose fundamental limits on achievable performance. Transformations are universally accepted as the most efficient way in overcoming these limitations.

We address throughput optimization using transformations and has two major goals. The first is to develop an approach for the maximally fast implementation of several important classes of computations with respect to eight important and powerful transformations (associativity, distributivity, commutativity, inverse element and zero element laws, common subexpression elimination and replication and constant propagation) can be efficiently derived. The second goal is to show how an arbitrarily fast (optimal with respect to the eight mentioned transformations and the amount of unfolding) hardware efficient implementation for all computations for which maximally fast implementation is achievable, can be produced. We also discuss how the new approach can be used for an increasingly important design parameter-power.

We start by summarizing the most relevant assumptions, abstractions, and definitions. A linear system [23] is a system L such that:

Manuscript received August 11, 1998; revised August 1, 1999. This paper was recommended by Associate Editor R. Gupta

M. Potkonjak is with the Department of Computer Science, University of California, Los Angeles, CA 90095 USA.

J. M. Rabaey is with the Department of EECS, University of California, Berkeley, CA 94720 USA.

Publisher Item Identifier S 0278-0070(00)01370-1.

1) if the response to a signal x is a signal y, then the response to ax is ay, where a is an arbitrary number (homogeneity or scaling property);
2) if the responses to x_1 and x_2 are y_1 and y_2, respectively, then the response to $x_1 + x_2$ is $y_1 + y_2$ (additivity property).

Linear computations use only addition, subtraction, and multiplication with constants as computational elements. Due to the superposition property (combination of homogeneity and additivity properties) of linear computations [23], it is possible to treat each output separately, and in the end superimpose (combine) them. This makes linear computations particularly amenable for the effective application of transformations. It is important to note that the definition of additions and multiplications is not necessarily restricted to their most widespread connotation in numerical computations, but is related to their role in algebraically specified structures. So all the results presented here are directly applicable to computations on groups where, for example, addition corresponds to max and multiplication to addition [9], or addition corresponds to the logical or and multiplication to the logical and.

The classification of computation into only two classes, linear and nonlinear, is for many purposes a coarse one. Therefore, we introduce a new class of computations, feedback linear. The motivation behind the introduction of the new class of computations is directly related to our main goal, to develop a transformation-based approach which will for as broad a class of computation as possible enable maximally fast and arbitrarily fast implementations.

Intuitively, the class of feedback linear computations encompasses all those computations which do not have multiplications (or divisions) between variables which belong to feedback cycles. More formally, feedback linear computations can be defined as those which can be described using:

$$X[n+1] = A(X)[n]) + B(U[n])$$

where $X[n]$ is a vector which denotes feedback states (algorithmic delays), $U[n]$ is a vector of primary inputs, and $Y[n]$ is a vector representation of primary outputs. A is a matrix whose entries are functions which do not depend on the feedback states $X[n]$. There is no restriction to the functions which form the entries of matrices B, C, and D. The importance of feedback linear computations is emphasized by the fact that many of modern very large scale integration (VLSI) applications belong to this class. The feedback linear class of computations includes Kalman, least mean square (LMS) adaptive, block

0278–0070/00$10.00 © 2000 IEEE

LMS, direct recursive least square (RLS) adaptive, Cholesky RLS adaptive, QR-RLS adaptive, and Volterra polynomial filters.

Identification of feedback linear computation is simple and can be done rapidly in the worst case quadratic time. The first step is the identification of cycles using the standard depth first search algorithm for the labeling of strongly connected components [5]. All nontrivial strongly connected components with more than one node, belong to cycles. All other nodes do not belong to cycles. The second step checks that all operations in cycles are additions, subtractions, and multiplications with either constants or with variables which do not have transitive fanout from some of feedback delays. During the second step, we assume that all inputs are constants. Standard constant propagation algorithms can be used for constant propagation [38], [31].

The assumed computational model is homogeneous synchronous data flow [18]. The data flow model specifies computation as a set of actors (nodes) which produce some number of samples to a subset of nodes only when they consume some number of samples (tokens) on each of their inputs. The synchronous data flow (SDF) model of computation is a special case of the data flow model of computation in which the number of data samples produced or consumed by each node on each invocation is specified *a priori*. The direct ramification is that nodes can be scheduled statically at compile time. This model is widely used in many application domains [digital signal processing (DSP), video and image processing, and communications]. Furthermore, there is an available set of execution units (e.g., adder, multiplier) so that each arithmetic or logic operation requires a certain fixed amount of time. The goal is to optimize the throughput of a computation using a set of transformations. The considered set of transformations encompasses distributivity, associativity, commutativity, inverse element law, zero element law, common subexpression elimination, common subexpression replication, and constant propagation [1].

Maximally fast implementation of an algorithm (or linear computation): is a functionally equivalent computational structure which can be obtained using the stated set of transformations and implemented using the available type of hardware in the minimal amount of time.

Arbitrarily fast implementation of linear computation: In this case the set of transformations is enlarged to include retiming and pipelining. Suppose that an arbitrary small amount of time T is specified. Arbitrary fast implementation of linear computation is a functionally equivalent computational structure to the initially specified computation which can be obtained using the stated set of transformations and implemented using the available type of hardware in the amount of time which is at most T.

Efficient implementation: Suppose that an initial computation can be implemented using area A_1 under timing constraint (throughput) T_1. If the computation is transformed that so that it can be implemented under timing constraint T_2 $(T_2 < T_1)$ using area A_2, the implementation is hardware efficient if $A_2 * T_2^2 < A_1 * T_1^2$.

Arbitrarily fast hardware efficient implementation: is efficient implementation under arbitrary strict timing constraints.

Finally, it is important to note that numerical stability is often greatly affected by the structure of a particular computation. We evaluate the impact of the proposed transformations scripts on numerical stability using the Hyper simulation tools [31] which compare in the time and frequency domain the results of simulation using double-precision floating point implementation and the specified fixed-point implementation. While in a few cases there has been either positive or negative minimal impact on the required wordlength, in a great majority of the designs there were no changes.

II. Prior Art

Transformations alter the structure of a computation in a such a way that the user specified input/output relationship is maintained. They are widely used for improving implementation of computations in a number of research and development domains, including compilers, VLSI design, computer arithmetic, theoretical computer science, logic synthesis, and high-level synthesis. The excellent references on the use of transformations in compilers is [1]. Therefore, we primarily concentrate our discussion of previous work on the use of transformations in computer-aided design (CAD), VLSI design and in particular high-level synthesis.

Common subexpression elimination is regularly used in many compilers and discussed in great detail in the compiler literature [1]. Most often the treatment of common subexpression in compiler research and development is based on value numbering [4] and Sethi–Ullman numbering [33] techniques in peephole optimization framework. While the use of common subexpression elimination is widely advocated and practiced in several behavioral synthesis systems [31], it seems that the only attempt of applying common subexpression replication can be attributed to Lobo and Pangrle [21].

While constant propagation can be solved straightforwardly when the computation does not involve conditionals, it can be easily shown that the constant propagation problem is undecidable in general [38]. The standard procedure to address this problem is to use conservative algorithms. Those algorithms do not guarantee that all constants will be detected, but that each data declared constant is indeed constant over all possible executions of the program.

Associativity, distributivity, and commutativity are the three most often used algebraic transformations. Most often they are treated under the paradigm of tree-height reduction, although several authors discuss in detail the critical path minimization of general directed acyclic graphs (DAG's) [21]. Recently, algebraic transformations have been generalized to cover other algebraic axioms such as inverse elements (e.g., $a-(b+c)=(a-b)-c$) and were applied with the goal of minimizing design area under throughput and latency constraints [27].

Other algebraic transformations are related to other axioms in algebraic structures, e.g., inverse and zero element law, and their potential as optimizing transformations, is in the early stage of exploration. Potkonjak and Rabaey combined power of associativity and inverse element to form a more powerful associativity transformation [27]. Iqbal *et al.* [13] used the zero inverse element law as part of algebraic speed-up procedure which trans-

forms computations under an arbitrary set of timing constraints on inputs and outputs using five algebraic transformations.

Several high-level synthesis publications discuss many important aspects of pipelining in depth. There is a very close relationship between retiming and pipelining [30]. Therefore, we will restrict our discussion to the retiming problem. When retiming is the only transformation of interest and the goal is the minimization of the critical path, several algorithms designed by Leiserson and Saxe provide an optimal solution [19]. When the goal is minimal area or power, the problem has been proven to be NP-complete [27]. There is one more important relationship between two transformations: pipelining and software pipelining [32], [11]. As described in [30], those two transformations represent the same computation structure alternation in two different computational models. While pipelining is applied on a semi-infinite stream of incoming data along the time loop, software pipelining is applied on a finite stream of data along the control loop. Loop unfolding (loop unrolling, loop unwinding) is one of those transformations which is regularly used in almost all optimizing compilers and many high-level synthesis systems [31].

We will conclude our survey of related research by itemizing transformation techniques which target specific instances of linear computations. From one standpoint, the work presented here can be described as a wide, implementation-efficient generalization of work done on the parallelizing linear recurrence $x_i = a_i + x_{i-1}$. In the last three decades, variations of linear recurrences under a variety of names, including parallel prefix computations, segmented scan, look-ahead, pipeline interleaving, and cyclic reduction, found an ubiquitous role in both theoretical and experimental computing.

In 1963, the fast computation technique for linear recurrences was used by Ofman to design a fast parallel circuit for the addition of binary numbers [22]. Since then, a great variety of this speed-up has been employed to design a variety of arithmetic computation structures, including carry-lookahead, carry-save, and Wallace tree multipliers. Ercegovac and Lang recently surveyed this line of research [7].

In 1967, a study of linear recurrence in the theoretical computer science community was initiated by Karp *et al.*[16]. Kung [17] calculated the bounds on performances and implementation complexity for parallel prefix circuits.

In 1968, the linear recurrences methodology was applied for the first time in the DSP domain [10]. Soon it was applied on a variety of signal processing problems [3], [37]. More recently, with the widespread use of VLSI in signal processing, this work was systemized and greatly enhanced, and successfully applied on several types of infinite impules response (IIR) filters and several classes of adaptive filters [9], [20]. In particular, we point out work done by Parhi and Messerschmitt which does not only summarize many of previous techniques and reduce their hardware overhead, but also introduces a number of new application domains [24]. Other important related work in the DSP domain include [9], [14], [15], [25] and [26].

It is instructive and important to compare the new technique with the related research. The maximally fast implementation of linear computation is a widely applicable topic which has not been studied previously. There is a number of techniques which provide arbitrarily fast implementations for various classes of computations [9], [10], [24], [37]. The new arbitrarily fast solution maintains asymptotically the AT product constant and, hence, is more hardware efficient than any of the previously proposed techniques. This is achieved using a novel Horner scheme-based pipelining technique which modifies the well know procedure for efficient calculation of polynomials to the new task of hardware efficient pipelining and is by itself an important tool for throughput and hardware optimization.

The treatment of the problem from a transformation standpoint provides several advantages. For instance, the methods presented here can be directly applied to the optimization of time loops in DSP arithmetic computations as well as control loops in software compilers. Also, it provides for the first time an arbitrarily fast implementation for a number of increasingly important new computational classes, such as nonlinear polynomial filters. It also indicates that the class of feedback linear computation covers all computations where arbitrary speed-up is achievable. Finally, the CAD approach to the problem provides a convenient way that an arbitrary linear or feedback linear computation is transformed to maximally and arbitrarily fast forms automatically. Until now, the state of art techniques required a significant creative and development effort for this task, and the resulting structures were regularly reported in journal publications [24], [9]. The CAD treatment provides a necessary link to the application of the proposed approaches to the optimization of a variety of new tasks. For instance, it enables power reduction by a factor of about 25 times in many linear and feedback linear computations [35].

III. New Approach: Idea and Introductory Examples

Probably the best way to introduce a basic idea is to use simple, yet nontrivial examples. Fig. 1 shows the control-data flow graph (SDF) of a three-port serial adaptor which is regularly used in many ladder digital filter structures, such as elliptic wave filters [8]. Since it is often a part of the feedback path of the complete filter and, therefore, cannot be pipelined; it usually dictates the overall throughput. Its critical path is denoted by the shaded nodes in Fig. 1 and equals seven clock cycles, assuming that each operation takes one cycle.

Iteration bound [6] imposes a fundamental limit on the highest throughput which can be achieved using an arbitrary level of pipelining and arbitrary retiming. Therefore, when a given computation is a part of a feedback loop, pipelining and retiming have limited effectiveness as critical path minimization techniques. Applying algebraic transformations (such as associativity) will only have limited success. Certain transformations which could lead to faster implementations are originally inhibited; for instance, associativity can only be applied when the intermediate variable does not have any extra fanout. Our goal now is to remove all those inhibiting factors first and, hence, extend the overall search space. This can be achieved in the following way.

Fig. 2 shows the same adaptor, however now the nodes which are in transitive fan-in of the output b_1 are shaded. In order to correctly compute this output it is sufficient to take into account

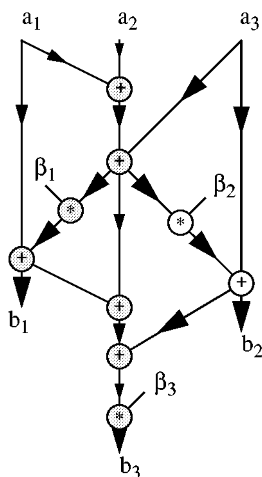

Fig. 1. SDF of a three-port serial adapter: shaded nodes belong to critical path.

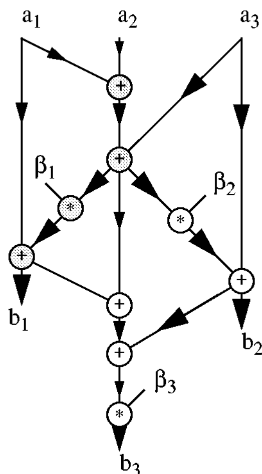

Fig. 2. SDF of a three-port serial adapter: shaded nodes denote the set of nodes on which output b_1 depends.

only this part of the computation. Fig. 3 shows the functional dependencies between all output nodes b_i and input nodes a_i. To get from the structure of Fig. 1 to the result of Fig. 3, we have effectively replicated all subexpressions common between the different output variables and, hence, made the computational trees independent.The resulting SDF has only additions and multiplications with constants as computational nodes. So, each output is a linear combination (weighted sum) of the inputs. Algebraic laws can now be effectively applied to generate the fastest possible solution.

Using this simple algebraic manipulation, followed by the use of constant propagation to precompute all input weights, the dependencies of Fig. 4 are obtained. The SDF of the restructured computation is presented in Fig. 5. It is easy to verify that the relationships between outputs b_i and inputs a_i are unaltered. However, the critical path is reduced to three clock cycles. The area of the resulting computational structure can finally be reduced by applying common subexpression elimination, distributivity and other algebraic transformations without increasing the critical path. For instance, note that when b_2 is computed we can reduce the required number of multiplications by one by applying distributivity $(\gamma_4 * a_1 + \gamma_4 * a_2 = \gamma_4 * (a_1 + a_2))$. The critical path is not altered.

If the application allows for the introduction of additional latency, the throughput can be increased even more by extending the transformational set. To convey the ideas behind the procedure, we will again use a small but real life example, as shown in Fig. 6. This figure shows the popular second-order IIR filter direct form II design, the standard building block in many recursive filters [23]. The filter has two feedback loops, which prevent the direct usage of pipelining or (time) loop unfolding. Fig. 7(a) shows the analytical expressions of the computations to be performed in a single iteration. A single unfolding of time loop (corresponding to the substitution of one equation into the next) results in the analytical expression of Fig. 7(b). The SDF of the unfolded IIR filter is shown in Fig. 8. The advantage of the new structure is that two iterations can be computed simultaneously. Unfortunately, the critical path of the unfolded structure is doubled, therefore, the throughput remains unchanged.

However, by transforming the graph using common subexpression replication, algebraic transformations, and constant propagation, the equations can be reorganized as shown in Fig. 7(c). Notice that the graph of Fig. 7(c) has exactly the same input/output relationship as the initial specification. The minimal execution time for this graph is only three clock cycles, which is identical to the critical path of the original graph of Fig. 6(a). As a result, the throughput has actually been doubled; two iterations can be performed in the time initially needed for one. Fig. 9 shows the computational structure of the unfolded implementation for the number of iterations.

Of course, we can continue with the loop unrolling and the subsequent reduction of the critical path using algebraic and redundancy manipulation transformations. The crucial observation is that regardless of the number of times the initial iteration is unfolded, each output will depend on only two initial states and as many inputs as the number of unfoldings. All primary inputs are out of feedback cycles and can be arbitrarily pipelined, so that the outputs depend only on one newly introduced pipeline delay. Therefore, the outputs and states of the unfolded computations will always depend on only three input variables. After organizing all these computations so that all the multiplications are done in the first cycle, followed by additions organized in a tree format, the critical path will stay unaltered, while we process more and more iterations. If the basic iteration is *unfolded* N times, the number of simultaneously processed samples and throughput will also increase N times. So an arbitrary fast implementation can be achieved by using the appropriate number of unrollings.

A natural and interesting question is simultaneous optimization of latency and throughput when linear computations are optimized. The techniques presented in this paper are used as a basis for answering this question, as presented in [34]. Srivastava and Potkonjak [34] presented a technique which provides a provably optimal combination of latency and throughput in a sense that any other feasible implementation of a given linear computation has either larger latency or smaller throughput. The techniques presented in this paper still have three important advantageous even when both latency and throughput are targeted during design: they are computationally more efficient, they re-

$$b_1 = \beta_1 * (a_1 + a_2 + a_3) + a_1$$

$$b_2 = \beta_2 * (a_1 + a_2 + a_3) + a_3$$

$$b_3 = \beta_3 * (\beta_1 * (a_1 + a_2 + a_3) + a_1 + \beta_2 * (a_1 + a_2 + a_3) + a_3 + (a_1 + a_2 + a_3))$$

Fig. 3. Functional dependencies between output and input nodes for the serial adaptor example.

$$b_1 = (\beta_1 + 1) * a_1 + \beta_1 * a_2 + \beta_1 * a_3$$

$$b_2 = \beta_2 * a_1 + \beta_2 * a_2 + (\beta_2 + 1) * a_3$$

$$b_3 = \beta_3 * (\beta_1 + \beta_2 + 2) * a_1 + \beta_3 * (\beta_1 + \beta_2 + 1) * a_2 + \beta_3 * (\beta_1 + \beta_2 + 2) * a_3$$

Fig. 4. Functional dependencies between output and input nodes for the serial adaptor example after the application of several transformations.

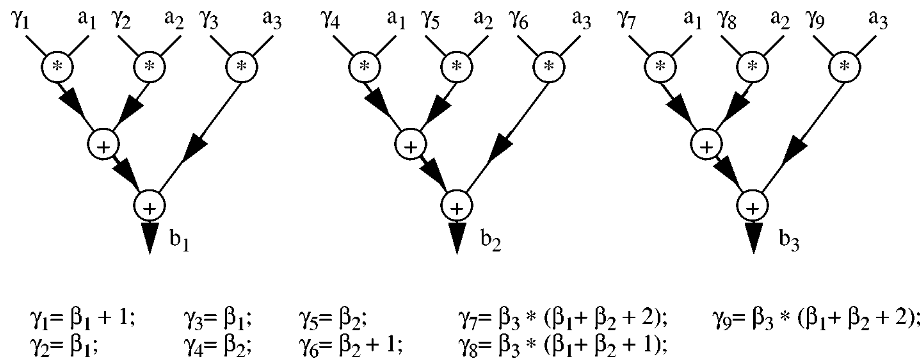

Fig. 5. SDF for restructured three-port serial adapter. Note that some of coefficients are 1, for those multiplications are not needed.

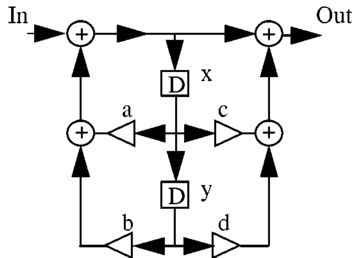

Fig. 6. Second-order IIR filter.

sult in more efficient implementations and they can be generalized to a wide class of computations.

IV. Optimizing Linear Systems

Due to dramatic improvements in silicon technology, design for minimal area has become less stringent in recent years. Even when the goal by itself is not sheer performance, it is often advantageous to have an option for improving the speed of design. Although the primary goal of the proposed algorithms is to maximize throughput, several additional factors cannot be ignored. Among them is the area of design, which is treated as the secondary cost in the analysis. In this section we will present the following three algorithms.

1) Maximally fast implementation of linear computations under latency constraints.
2) Arbitrarily fast implementation of linear computations.
3) For minimization of the number of multiplications and additions in linear computations.

We already introduced the critical path minimization algorithm. It can be formalized with the following pseudocode.

Algorithm for Maximally Fast implementation of Linear Computations:

1) Express each output node as linear combination of the input nodes;
2) Group and then pipeline all primary inputs, so that each output (either primary or algorithmic delay) depend on only one new pipeline delay;
3) Arrange the SDF for computing each output node using the fastest tree structure;
4) Minimize the number of operation in trees by sharing common subexpressions in trees for different outputs.

The first step of the algorithm uses common subexpression replication as the enabling transformation for the critical path minimization, which is conducted during the third step. The relationship between an output node and the input nodes can be obtained in polynomial time using dynamic programming. An even simpler and more elegant approach can be built for this task by exploring the properties of linear computations. We assign a set of proper values to the input nodes and compute the values of the output nodes in the SDF. In order to figure out how all output nodes depend on one particular input node, we will utilize the fact that there is a linear relationship between each input and output node in the following way. First, the value one is assigned to this input node and the value zero to all other inputs nodes.

The second step, pipelining of primary inputs, is an enabling transformation which greatly reduces the influence of primary

First Iteration:

$$x_1 = In_1 + a*x_0 + b*y_0$$

$$y_1 = x_0 \qquad \text{(a)}$$

$$Out_1 = x_1 + c*x_0 + d*y_0$$

Second Iteration:

$$x_2 = In_2 + a*x_1 + b*y_1$$

$$y_2 = x_1 \qquad \text{(b)}$$

$$Out_2 = x_2 + c*x_1 + d*y_1$$

After Unrolling and Algebraic Transformations:

$$x_2 = In_2 + a*In_1 + (a*a + b)x_0 + a*b*y_0$$

$$y_2 = In_1 + a*x_0 + b*y_0 \qquad \text{(c)}$$

$$Out_1 = In_1 + (a + c)x_0 + (b + d)y_0$$

$$Out_2 = In_2 + (a + c)In_1 + (a*a + b + c*a + d)x_0 + (a*b + c*b)y_0$$

Fig. 7. Functional dependencies for second-order IIR filter: (a) and (b) first two iterations before and (c) after unfolding.

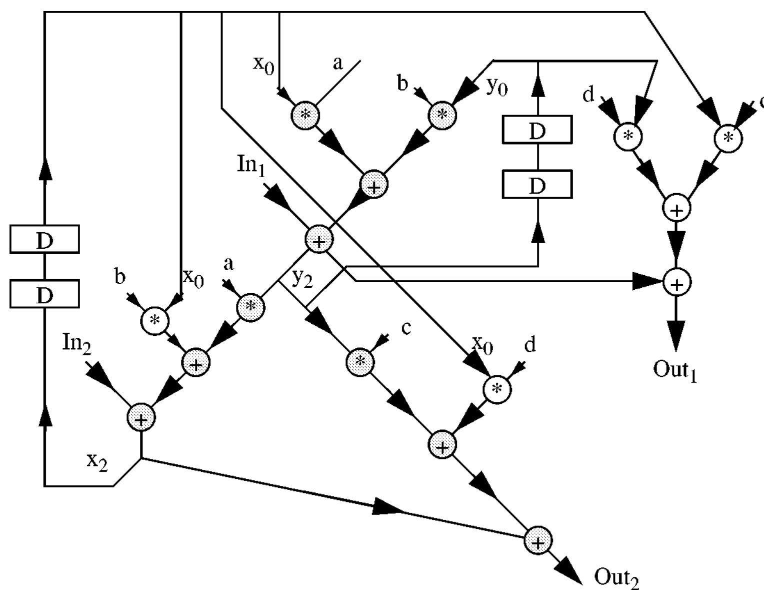

Fig. 8. SDF of the unfolded second-order direct form-II IIR filter.

inputs. If the SDF has only one primary input and all outputs have a weight one in the linear dependency equations which express the relationship between the outputs and this input, then this step can be skipped. This step is the most effective when the design has a large number of primary inputs and relatively few algorithmic delays in feedback cycles.

The third step is the well-known transformation from chain of additions to binary tree structure using associativity, which often sharply reduces critical path. The last step does not influence the length of the critical path. It is used for the optimization of the secondary goal, area. From the above analysis, it can be easily derived that the run-time of the algorithm is at most quadratic in the number of nodes.

Theorem 1 establishes the important property that the transformed design, as obtained from the proposed algorithm using the set of all algebraic transformations, common subexpression replication and elimination, constant propagation, and pipelining, is the fastest possible. The factor N, used in the formulation, denotes the number of delays in feedback cycles.

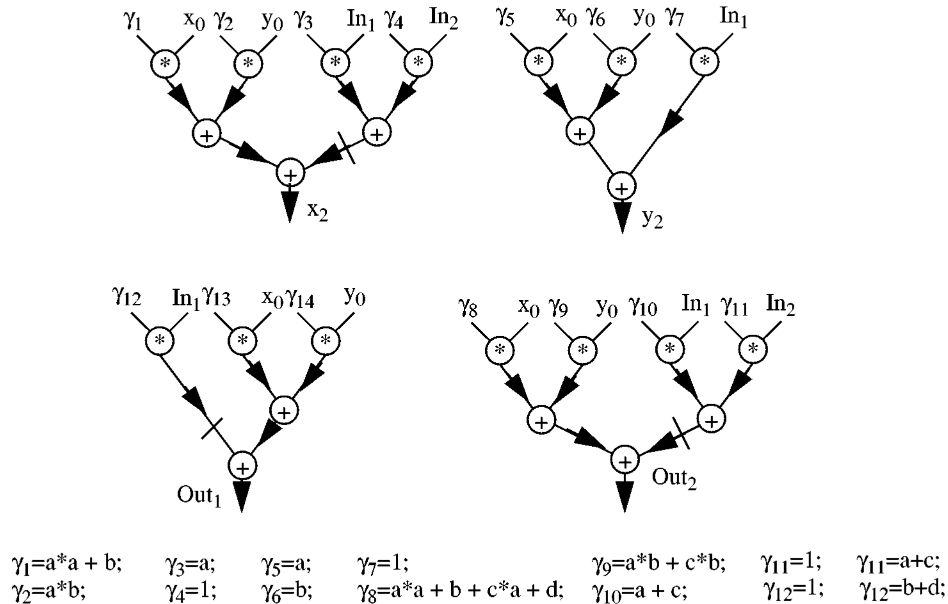

Fig. 9. Computational structure of the unfolded and restructured Implementation of the second-order IIR filter. Crossed edges denote positions of pipelining delays.

Theorem 1: Given a computational graph of a linear computation, the fastest implementation (obtained using only algebraic transformations, common subexpression elimination and replication, constant propagation, and pipelining of primary inputs) uses at least $\lceil \log N + 1 \rceil$ computational levels. Therefore, the implementation provided by the proposed algorithm is maximally fast.

Proof: We can draw this conclusion by observing that in order to compute a particular output using two input functional elements, we need at least $\log N$ levels of computational elements, since at each level the number of fan-ins is reduced by a factor of 2. We need one more level for multiplications by constants. The proposed algorithm reaches this lower bound.

When in addition to the above set of transformations, other transformations such as retiming are allowed, the speed of the final implementation can be even further improved. For instance, retiming combined with other transformations can be used to minimize the maximum number of algorithmic delay inputs on which an output depends. Currently, we use the probabilistic sampling algorithm [27] for this complex task. It is important to note that in some cases further improvements are achievable if a large set of transformation is simultaneously considered. For example, it has been demonstrated [14] how the constant/expression factoring can be used an ascent/steepest descent search for additional area and power optimization of SDF.

When the introduction of additional latency is allowed, we can additionally reduce the critical path by trading latency for throughput. In this subsection, we will show how the algorithm presented in the previous subsection, can be used as a starting point for developing an efficient technique for the arbitrary speedup of linear computation within the limitations of physical wiring and internal transistor switching.

We start by observing that an arbitrary linear computation can be readily transformed, using the algorithm for maximally fast implementation of linear computations, so that the SDF dependencies can be expressed in the following form:

$$X[n+1] = A * X[n] + B * U[n] \ @ \ 1.$$

Sign @ denotes the introduction of pipeline stages. All four matrices A, B, C, and D have only constants as entries. The part of SDF for computing primary outputs, outside of cycles, can be pipelined. Therefore, the throughput is dictated only by matrix A. Similarly, as the common subexpression replication was enabling transformation for maximally fast implementation of a linear computation, in order to achieve arbitrarily fast implementation it is necessary to initially apply the enabling transformation. For this task we will use time loop unfolding. Note that we not only need to compute all the corresponding outputs, but also a new state, so that the computations can be correctly continued when a new set of input data arrives. The key observation, which directly leads to the conclusion that the arbitrary speed-up is feasible, is that each output and all delays depend only linearly on the delays and inputs. The application of the maximally fast algorithm on the set of equations directly indicates that regardless of the how many iterations we implement simultaneously, the critical path of the unfolded and transformed structure is not altered. Therefore, an efficient algorithm for arbitrarily fast implementation of linear computations problem can be described using the following pseudocode:

Algorithm for Arbitrarily Fast implementation of Linear Computations:

1) Retime data control flowgraph so that the number of delay elements in the SDF is minimized or the maximum number of inputs on which some output depends is minimized;
2) Unfold the basic body of the iteration N times (as dictated by latency, hardware or other user-defined constraints).

3) Using the algorithm for maximally fast implementation of linear computations, minimize the critical path in the unfolded structure.

After the unfolding of the basic iteration by the factor N, each output in the transformed SDF will linearly depend on some of the inputs in N iterations and one pipeline delay. Of course, it is advantageous for the eventual throughput if the outputs (either primary outputs or delays) depend on as few as possible inputs from delays. We propose two different preprocessing options for this tasks. The first is a restricted, sometimes suboptimal, strategy where only retiming is considered. This approach is associated with the optimization problem which can be solved in polynomial time optimally. The second strategy is globally more optimal.

The first alternative is based on the premise that if the number of delays is minimized the chances for a high-quality solution are increased. The number of delays can be reduced using retiming. This step (retiming for the minimization of the number of delays) can be solved in $O(|E|^2 \log |V| + |V||E| \log^2 |V|)$ time by reducing the problem to the minimum-cost problem and solving it using the Orlin algorithm [19] (where $|V|$ is the number of operational elements, and $|E|$ is the number of edges in the SDF). Since the number of edges is linear to the number of nodes, the running time is essentially quadratic. Sometimes, even better results (solutions with fewer delays) can be obtained if the power of retiming is combined with algebraic and other transformations. However in this case the optimization problem becomes more computationally difficult, and no polynomial complexity algorithm using transformations is available. We again use for this case the probabilistic sampling algorithm [27]. Note that the run-time of the algorithm is dominated by this step.

The second alternative to this preprocessing step is based on the following observation. Each output after an arbitrarily large number of unfolding will depend only on delays which are in the transitive fanout of that delay. The transitive fanout has to be taken over several time frames (iterations). Restructuring of the targeted SDF using a variety of transformations so that the maximum number of delays on which any of the outputs depend is a complex optimization problem. Since, even the computation of the objective function takes significant time in this problem, we are currently using the first alternative.

The second step, unfolding of the initial iteration over the time loop, is straightforward and no optimization is involved. The final step is the direct application of the algorithm described in the previous subsection. So we can conclude that the run time of the algorithm for arbitrarily fast implementation of linear computation is also quadratic.

Although three types of lower bounds on the area and time of VLSI designs have been proposed (A, AT and AT^2), almost all strong lower bounds that do match the best circuits which can be constructed are lower bounds on the product AT^2 [36]. We will conclude this subsection with two theorems which clearly indicate effectiveness and efficiency of the algorithm for arbitrarily fast implementation of linear computations. The first theorem states that not just AT^2 product, but also AT product is maintained, regardless of the requested speed-up.

Theorem 2: The ratio of the initial and the final AT product of the designs produced using the algorithm arbitrarily fast

$$
\begin{aligned}
Y_1 &= CX_1 + DU_1 \\
Y_2 &= CAX_1 + C(BU_1) + DU_2 \\
Y_3 &= CA^2X_1 + CABU_1 + CBU_2 + DU_3 \\
&\vdots \\
Y_n &= CA^{n-1}X_1 + C(A(\qquad)) + DU_{n-1}
\end{aligned}
$$

Fig. 10. Application of generalized Horner scheme on the arbitrarily fast implementation of an arbitrary linear computation.

implementation of linear program is constant for an arbitrary throughput improvement.

Proof: The proof is constructive. It is based on the novel use of Horner's rule for polynomial evaluation. The rule rearranges the computation of an nth degree polynomial

$$u(x) = u_n x^n + u_{n-1} x^{n-1} + \cdots + u_1 x + u_0, \qquad u_n \neq 0$$

to the following form:

$$u(x) = (\cdots (u_n x + u_{n-1})x + \cdots)x + u_0.$$

Therefore, an arbitrary polynomial can be computed using at most n additions and n multiplications.

A simple analysis shows that direct implementation of the targeted linear computation after the application of the algorithm presented in this subsection causes the resources to grow at a quadratic pace, as the function of the number of unfoldings. We apply the key idea from Horner's scheme, on the part of the computation used to compute the influence of primary inputs, so that this overhead is reduced to linear increase. The resulting structure is shown in Fig. 10 using the functional dependency form. Note that we can add to this computational structure as many pipeline delays as requested. Also note that with any additional level of unfolding only the constant amount of computation is added: what is within a constant factor equal to how much more computation is needed if the new speed-up algorithm is not applied. Therefore, the AT product is constant.

Theorem 3: When the set of transformations is restricted to algebraic transformations, loop unfolding, pipelining, and redundancy manipulation techniques, the arbitrarily fast algorithm produces designs with the highest throughput for a given latency.

Proof: Once again the proof can be easily established by observing that eventually every output will depend on one algorithmic delay used to pipeline all primary inputs and all delays in its transitive fanout. Using the same limited fanin argument as in Theorem 1, the correctness of the theorem can be easily established. Q.E.D.

While targeting speed as a primary goal, keeping the overall area of the design as small as possible is definitely important as well. The additional use of common subexpression elimination, distributivity, commutativity, and reduction in strength, can reduce the number of multiplications and additions/subtractions significantly. As a result, the final implementation often does not have only high throughput, but also small area. An additional benefit is that by reducing the number of operations, we also often reduce the number of switching events. Therefore, the algorithms presented in this subsection significantly enhance the

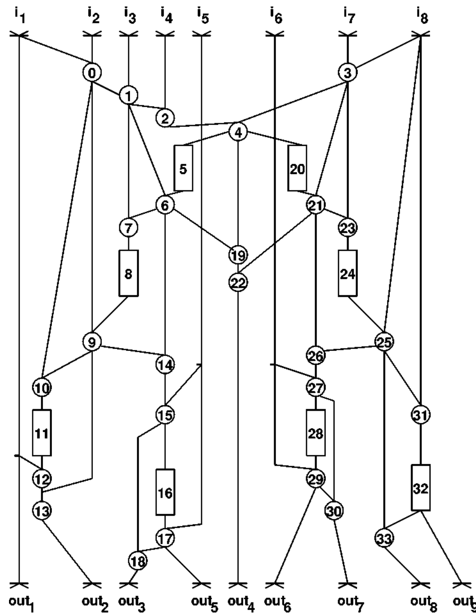

Fig. 11. Fifth order elliptical wave digital filter. Rectangles denote two cycle long multiplications, circles denote unit cycle additions.

application domain of maximally fast and arbitrarily fast techniques and makes them an attractive option for producing area and power competitive designs.

We will use a fifth-order wave digital elliptical filter as an illustrative example to introduce both the problem formulation and the algorithm for solving this new optimization task. Fig. 11 shows the computational graph of this popular benchmark. There are eight inputs for one iteration of the filter (seven delays and one primary input i_1) and eight outputs (seven delays and one primary output o_9). The entries in Table I show the functional dependencies between outputs and inputs (assuming that constant weighting factors for multiplications 5, 20, 8, 24, 11, 16, 28, and 32 (see Fig. 6) are set to the values of 2, 3, 4, 5, 6, 7, 8, and 9, respectively[1]). Table I and Fig. 11 show for each output O_i the inputs I_i on which it depends. By counting those dependencies, we can conclude that 47 multiplications and 39 additions are needed. A few multiplications can be eliminated, as they represent algebraic identities. For instance, multiplications with constant 1 can be eliminated by invoking the identity element law.

Fig. 12 shows the same functional dependencies after the application of distributivity for all inputs, which are multiplied with the same coefficient in the final implementation. Also note that several intermediate sums (e.g., $i_1+i_2+i_3+i_4$ in the outputs o_6, o_7, and o_8) are the same and have to be computed only once by invoking common subexpression elimination.

Those two observations form a basis for a relatively simple, yet effective and efficient algorithm which first minimizes the number of multiplications using distributivity for each output, and then uses common subexpression elimination on the intermediate result for the different outputs. The algorithm significantly reduces the number of operations in the transformed design. We target primarily the minimization of the number of multiplications, because they are significantly more expensive than the additions for fixed-point computations which dominate the application domain of linear computations.

[1]Those values were chosen for the sake of demonstration only and have no physical meaning.

TABLE I
FUNCTIONAL DEPENDENCES BETWEEN INPUTS AND OUTPUTS FOR Fifth-Order ELLIPTICAL FILTER

	$\mathbf{i_1}$	$\mathbf{i_2}$	$\mathbf{i_3}$	$\mathbf{i_4}$	$\mathbf{i_5}$	$\mathbf{i_6}$	$\mathbf{i_7}$	$\mathbf{i_8}$
$\mathbf{o_1}$	1	0	0	0	0	0	0	0
$\mathbf{o_2}$	126	125	112	56	0	0	56	56
$\mathbf{o_3}$	160	160	152	80	9	0	80	80
$\mathbf{o_4}$	7	7	7	6	0	0	7	7
$\mathbf{o_5}$	140	140	133	70	8	0	70	70
$\mathbf{o_6}$	144	144	144	144	0	9	232	240
$\mathbf{o_7}$	162	162	162	162	0	10	261	270
$\mathbf{o_8}$	150	150	150	150	0	0	250	269
$\mathbf{o_9}$	135	135	135	135	0	0	225	243

The algorithm first applies distributivity on all input variables for computing each output, so that as many multiplications with the same constants are eliminated. For example, during the computation of $\boldsymbol{o_7}$ the application of distributivity, augmented with commutativity, ($\boldsymbol{o_5} = 140*i_1+140*i_2+133*i_3+70*i_4+8*i_5+70*i_7+70*i_8 = 140*(i_1+i_2)+133*i_3+70*(i_4+i_7+i_8)+8*i_5$) reduces the number of multiplications by four.

The next step is the reduction of multiplication based on exploring common subexpression elimination. The algorithm looks for all variables which are weighted by the same coefficient in different outputs. Obviously, it is sufficient to compute the product of those input variables with those coefficients only once. In the demonstration example, the previous step by itself does not reduce number of multiplications.

The third step is the application of reduction in strength [1]. Reduction in strength is a popular compiler transformation which substitutes a more expensive operation with a less expensive one. Most often multiplications are replaced by additions. We apply reduction in strength in two phases. In the first phase, all variables which are multiplied by pairs of constants of type $(c, c+1)$, where c is an arbitrary constant in various linear combinations which are targeted. There are three such cases in the fifth-order filter (i_5 in $\boldsymbol{o_3}$ and $\boldsymbol{o_5}$; i_6 in $\boldsymbol{o_6}$ and $\boldsymbol{o_7}$, and i_8 in $\boldsymbol{o_7}$ and $\boldsymbol{o_8}$). We trade one multiplication for one addition by replacing computations $\text{int}_1 = c*i_i$ and $\text{int}_2 = (c+1)i_i$ by $\text{int}_1 = c*i_i$ and $\text{int}_2 = \text{int}_1+1$. In the second phase we identify pairs of computations used for computing any single output of type $c*i_k+(c+1)*i_j$ and replace it by $c*(i_k+i_j)+i_j$. Again, we trade one multiplication for one addition. One instance of this tradeoff is $\boldsymbol{o_2}$. Here, subexpression $(126*i_1+125*i_2)$ is replaced by $125*(i_1+i_2)+i_1$ and, therefore, we replaced one multiplication with one addition. Note that the effectiveness of strength reduction can vary a lot depending on a particular set of coefficients in a linear or feedback-linear computation.

The final step is the minimization of the number of additions using common subexpression elimination. Note, for example, that the expression $(i_1+i_2+i_3+i_4)$ is used in calculation of $\boldsymbol{o_6}, \boldsymbol{o_7}$, and $\boldsymbol{o_8}$. It is, of course, sufficient to calculate it only once. Also note that during the computation of $\boldsymbol{o_2}, \boldsymbol{o_3}$, and $\boldsymbol{o_5}$ it is advantageous to form an intermediate result $i_4+i_7+i_8$,

$$\mathbf{o_1} = 1 * i_1;$$

$$\mathbf{o_2} = 126 * i_1 + 125 * i_2 + 112 * i_3 + 56 * i_4 + 56 * i_7 + 56 * i_8;$$

$$\mathbf{o_3} = 160 * i_1 + 160 * i_2 + 152 * i_3 + 80 * i_4 + 9 * i_5 + 80 * i_7 + 80 * i_8;$$

$$\mathbf{o_4} = 7 * i_1 + 7 * i_2 + 7 * i_3 + 6 * i_4 + 7 * i_7 + 7 * i_8;$$

$$\mathbf{o_5} = 140 * i_1 + 140 * i_2 + 133 * i_3 + 70 * i_4 + 8 * i_5 + 70 * i_7 + 70 * i_8;$$

$$\mathbf{o_6} = 144 * i_1 + 144 * i_2 + 144 * i_3 + 144 * i_4 + 9 * i_6 + 232 * i_7 + 240 * i_8;$$

$$\mathbf{o_7} = 162 * i_1 + 162 * i_2 + 162 * i_3 + 162 * i_4 + 10 * i_6 + 261 * i_7 + 270 * i_8;$$

$$\mathbf{o_8} = 150 * i_1 + 150 * i_2 + 150 * i_3 + 150 * i_4 + 250 * i_7 + 269 * i_8$$

$$\mathbf{o_9} = 135 * i_1 + 135 * i_2 + 135 * i_3 + 135 * i_4 + 225 * i_7 + 243 * i_8$$

Fig. 12. Functional dependencies of the fifth-order elliptical wave digital filter.

$$im_{12} = i_1 + i_2; \quad im_{34} = i_3 + i_4;$$

$$im_{1234} = i_{12} + i_{34}; \quad im_{78} = i_7 + i_8;$$

$$im_{478} = i_4 + i_{78}; \quad im_5 = 8 * i_5;$$

$$im_{5s} = im_5 + 1; \quad im_6 = 9 * i_6;$$

$$im_{6s} = im_6 + 1; \quad im_8 = 269 * i_8;$$

$$im_{8s} = im_8 + 1; \quad im_{12m} = 125 * im_{12};$$

$$im_{12s} = im_{12m} + 1;$$

$$O_2 = im_{12s} + 112 * i_3 + 56 * im_{78};$$

$$O_3 = 160 * im_{12} + 152 * i_3 + 80 * im_{478} + im_{5s};$$

$$O_4 = 7 * (im_{1234} + im_{78}) - i_4;$$

$$O_5 = 140 * im_{12} + 133 * i_3 + 70 * im_{478} + im_5;$$

$$O_6 = 144 * im_{1234} + im_6 + 232 * i_7 + 240 * im_{8s};$$

$$O_7 = 162 * im_{1234} + im_{6s} + 261 * i_7 + im_{8s};$$

$$O_8 = 150 * im_{1234} + 250 * i_7 + im_{8s};$$

$$O_9 = 135 * im_{1234} + 225 * i_7 + 243 * im_{8s};$$

Fig. 13. The SDF of the fifth-order elliptical wave digital filter after the application of the maximally fast procedure and the procedure for the minimization of the number of operations.

while when $\boldsymbol{o_6}, \boldsymbol{o_7}$, and $\boldsymbol{o_8}$ are computed the most suitable intermediate result is $i_1 + i_2 + i_3 + i_4$. So, in the general case it is not clear which intermediate sums to form so that the number of additions is minimized. This observation indicates that the minimization of the number of addition in the fast implementations of linear computation is an involved optimization problem. We use for this task the recently developed iterative pairwise matching algorithm for the minimization of the number of additions in multiple constant multiplications [28].

So, the algorithm for the minimization of the number of multiplications and additions in linear maximally fast or arbitrarily fast computations can be expressed using the following pseudocode.

Algorithm for Minimization of the Number of Operations:

1) Minimize the number of multiplications within each output by applying distributivity.
2) Minimize the number of multiplications by using common subexpression elimination targeting input variables weighted by the same factors for calculation of the same output.
3) Substitute multiplication for addition using strength reduction and common subexpression elimination.
4) Minimize the number of additions using multiple constant multiplication algorithm [28].

The computational complexity of the algorithm is dominated by Step 4. For this step, we use the MCM algorithm [28] which has quadratic complexity. After the application of each step of the algorithm for minimization of the number of operation, the length of the resulting critical path is calculated, and the transformation is revoked if the new critical path exceed the initial critical path achieved using arbitrarily or maximally fast techniques.

Fig. 13 shows the final SDF for efficient calculation of the maximally fast fifth-order elliptical wave digital filter. For the fifth-order elliptical filter example, only 23 multiplications and 29 additions are finally needed for a critical path of five cycles (assuming that a multiplication takes two cycles). The original implementation uses eight multiplications and 26 additions and has a critical path of 17 cycles. Both the number of additions and the number of multiplication increased but at a slower rate then the reduction in the critical path. Therefore, although our primary goal was the throughput, both AT and AT^2 products are improved for this example. When the available time is 12 cycles, the proposed computational graph needs only four mul-

tipliers and three adders, which is three adders less than in the previously best published implementation [21] for this example.

There is one more simple, but often effective optimization step which does not alter the number of operations, but often significantly reduces the number of execution units in the final solution of fast implementations of linear computations. It is based on the observation that in both maximally fast and arbitrarily fast implementations, all multiplications are bound to be scheduled at the beginning of an iteration, and all additions are bound to be scheduled at the end of the iteration. For the operations in the feedback cycles often there is no room for properly spreading operation of the same type through time, which is a prerequisite for high resource utilization of execution units [29]. However, for all operations which are out of the feedback cycles (i.e., those which depend only on the primary outputs and those which are used for the calculation of the primary outputs), the application of pipelining always ensures an arbitrary positioning of operations within the iteration.

V. Optimizing Feedback Linear Computations

In this section we present how the fast throughput techniques from the previous section can be applied, with minor modifications, so that any computation which belongs to a wide and important class of feedback linear computations can be transformed so that its throughput is maximally increased. The modifications are based a simple, but far reaching observations and one well-know result from theoretical computer science. The observation is that the proper approach to effective application of transformations has as the first step the identification of bottlenecks in the computational structure which cannot be handled easily. When the target is throughput, the only possible bottlenecks are feedback cycles. Recall that all operations out of feedback cycles can always be readily pipelined to any requested level.

All theoretical and algorithmic results from the previous section can be easily adapted for the feedback linear system. The only difference is that while previously we treated A, B, C, and D as matrices with constant entries, now they represent vector operators. It is important to note that the new semantic meaning associated with the four matrices has a limited, but significant impact on hardware overhead. The entries of operator A are now symbolic expressions and, therefore, constant propagation cannot be applied in general case. Therefore, each additional application of operator A, induces in the worst case quadratic overhead.

Before finishing this section by stating three main theorems about the throughput of an arbitrary feedback linear computation, we introduce a definition of constant feedback linear computations. Informally, constant feedback linear computations are feedback linear computations which have only additions, subtractions, and constant multiplications in feedback cycles. A more formal definition is that constant feedback computations are computations which can be represented in form

$$X[n+1] = A * X[n] + B * (U[n])$$

While Matrix A has only constants as entries, there are no restrictions on matrices B, C, and D.

Theorem 4: Given a computational graph of a feedback linear computation, the fastest implementation of this computation (obtained using only algebraic transformations, common subexpression elimination and replication, constant propagation, and pipelining of primary inputs) uses at least $\log N + 1$ computational levels and is provided by the algorithm presented in Section IV-A.

Theorem 5: The ratio of the initial and the final AT^2 product of the designs produced using the algorithm arbitrarily fast implementation of feedback linear program is constant for an arbitrary throughput improvement.

Theorem 6: The ratio of the initial and the final AT product of the designs produced using the algorithm arbitrarily fast implementation of constant feedback linear program is constant for an arbitrary throughput improvement.

Of course, the last two theorems are correct only for large number of unfoldings. Even then the intractability of physical designs problems and imperfections of used models and synthesis tools will have some (sometimes large) impact. Proof of the last theorem can be readily established by noting that in this case matrix A^n can be precomputed using constant propagation techniques.

VI. Experimental Results

The algorithms presented here, except for optional preprocessing retiming and postprocessing minimization of the number of used operation steps, are of polynomial complexity (actually most often quadratic) and guarantee the optimal solutions. So the major question is not the validation of algorithms, but how much improvement can be achieved using the concept and CAD framework and what is the associated hardware overhead.

The critical path before and after the application of the algorithm for the maximally fast implementation of linear computations on a number of benchmarks is shown in Table II. The examples are: mat1–3 state GE controller, lin4–5 state GE controller, steam–steam power controller, chemical–chemical plant controller, dist-distillation plant controller; 5IIR–fifth-order elliptical wave digital filter, 7IIR–seventh-order IIR cascade filter, 8IIR–eighth-order Avenhaus filter, 11–eleventh-order IIR Chebishev filter, 12IIR–twelfth-order Butterworth filter, 18IIR–eighteenth-order IIIR filter, 22IIR–twenty-second-order IIR filter, and DAC–NEC digital-to-analog converter. The average and median improvement in throughput are by factors 3.92 and 2.45, while the average and median increase in area are by factors 2.60 and 1.75, respectively.

The same example was applied on four feedback linear examples: two nonlinear fixed coefficients Volterra filters and two LMS adaptive filters. For this group of examples, the average and median improvement in throughput are 4.25 and 4.00 times, while the average and median increase in area are 2.58 and 2.50 times, respectively (Table III).

Two examples, lin4 and fourth-order LMS adaptive filter were studied for analysis of arbitrarily fast algorithm. The behavior of the two examples was drastically different. While for the linear computation example, overhead in both power and area were modest, the feedback linear example experienced very high area overhead. For example, when the throughput of

TABLE II
FAST IMPLEMENTATION OF LINEAR DESIGNS: ICP/FCP—INITIAL/FINAL CRITICAL PATH; IFCP—ICP/FCP RATIO; IA/FA—INITIAL/FINAL AREA; FIA—IA/FA RATIO; IFAT—RATIO OF THE INITIAL AND THE FINAL AT PRODUCT

Design	ICP	FCP	IFCP	IA	FIA	AIFA	IFAT
mat1	5	3	1.67	6.62	7.96	1.20	1.39
lin4	6	4	1.50	13.01	15.98	1.23	1.22
steam	7	4	1.75	27.01	46.75	1.73	1.01
dist	7	4	1.75	12.18	13.47	1.11	1.58
chemical	6	3	2.00	14.97	25.56	1.71	1.17
5WDF	17	5	3.40	21.92	81.55	3.72	0.91
7IIR	10	4	2.50	27.11	42.58	1.57	1.59
8IIR	11	5	2.20	29.25	49.90	1.71	1.29
10IIR	12	5	2.40	31.31	55.42	1.77	1.36
11IIR	17	5	3.40	36.50	66.26	1.82	1.87
12IIR	20	5	4.00	49.27	172.45	3.50	1.14
18IIR	29	6	4.83	56.60	233.97	4.13	1.17
22IIR	25	6	4.17	69.81	238.01	3.40	1.23
DAC	58	3	19.33	5.48	42.99	7.84	2.47

TABLE III
FAST IMPLEMENTATION OF NON-LINEAR DESIGNS: ICP/FCP—INITIAL/FINAL CRITICAL PATH; IFCP—ICP/FCP RATIO; IA/FA—INITIAL/FINAL AREA; FIA—IA/FA RATIO; IFAT—RATIO OF THE INITIAL AND THE FINAL AT PRODUCT

Design	ICP	FCP	IFCP	IA	FA	FIA	IFAT
2 Volterra	12	2	6.00	28.86	94.01	3.26	1.84
Volterra	8	2	4.00	27.86	55.21	1.98	2.02
4 LMS	12	4	3.00	55.75	134.97	2.42	1.24
8 LMS	20	5	4.00	111.15	298.03	2.68	1.49

TABLE IV
APPLICATION OF ARBITRARILY FAST PROCEDURE ON FIFTH-ORDER LINEAR CONTROLLER: CP—EFFECTIVE CRITICAL PATH FOR ONE ITERATION; ENERGY—ENERGY AT 5-V PER SAMPLE; ENERGYS—ENERGY AT SCALED VOLTAGE PER SAMPLE; CPR, AREAR, ENERGYR, ENERGYSR—RATIO BETWEEN INITIAL AND FINAL CRITICAL PATH, AREA, ENERGY AT 5 V, AND ENERGY AT THE SCALED VOLTAGE PER SAMPLE, RESPECTIVELY. THE COLUMNS REPRESENT INITIAL DESIGN AND DESIGNS AFTER APPLICATION OF n TIMES UNFOLDED ARBITRARILY FAST PROCEDURE

Parameter	In	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
CP	6	4	2	1.33	1	0.80	0.67	0.57	0.50	0.44	0.40
Area	13.0	16.0	21.2	26.6	30.6	33.3	35.8	40.4	48.2	50.9	60.3
Energy	55.7	63.2	67.8	110	120	141	149	167	190	212	253
EnergyS	55.7	27.5	12.0	13.8	12.3	12.7	12.3	12.5	13.5	14.0	15.7
CPR	1.00	1.50	3.00	4.50	6.00	7.50	9.00	10.5	12.0	13.5	15.0
AreaR	1.00	1.23	1.63	2.05	2.35	2.56	2.75	3.11	3.71	3.77	4.64
EnergyR	1.00	1.13	1.22	1.97	2.15	2.53	2.68	3.00	3.41	3.81	4.54
EnergySR	1.00	0.49	0.22	0.25	0.22	0.23	0.22	0.22	0.24	0.25	0.28

the fifth-order controller was increased 15 times, area increased only 4.6 times, and the used energy per sample increased only 4.5 times. If we utilize the increase in throughput to reduce lower supply voltage, then Table IV indicates that power can be reduced by a factor of almost five. All power estimates indicates that power can be reduced by a factor of almost five. All power estimates were obtained using the statistical Hyper-LP power model, which takes into account all components of both datapath and control path, including execution units, registers, interconnect, and control logic [2]. However, for the feedback linear example (Table V) the area overhead was more than 12 times when throughput is increased six times. Similarly, the effective capacitance increased at an even higher pace. Interestingly, still it was advantageous to apply voltage scaling, and trade improved throughput for energy saving.

Note that if a goal is area or power minimization for a given throughput, the proposed technique are not necessarily optimal. For further discussion on area and power minimization using transformations, recommended references include [12].

VII. CONCLUSION

Using the relationship between the superposition property for linear computations and the set of optimizing transformations,

TABLE V
APPLICATION OF ARBITRARILY FAST PROCEDURE ON VOLTERRA FILTER: CP—EFFECTIVE CRITICAL PATH FOR ONE ITERATION; ENERGYS—ENERGY AT 5-V PER SAMPLE; ENERGY—ENERGY AT SCALED VOLTAGE PER SAMPLE; CPR, AREAR, ENERGYR, ENERGYSR—RATIO BETWEEN INITIAL AND FINAL CRITICAL PATH, AREA, ENERGY AT 5 V, AND ENERGY AT THE SCALED VOLTAGE PER SAMPLE, RESPECTIVELY. THE COLUMNS REPRESENT INITIAL DESIGN AND DESIGNS AFTER APPLICATION OF n TIMES UNFOLDED ARBITRARILY FAST PROCEDURE

Parameter	In	A1	A2	A3
CP	6	3	1.5	1
Area	26.28	56.27	103.97	317.56
Energy	98.0	207	401	1239
EnergyS	98.0	62.6	54.3	128
CPR	1.00	2.00	4.00	6.00
AreaR	1.00	2.14	3.96	12.1
EnergyR	1.00	2.11	4.09	12.6
EnergySR	1.00	0.64	0.55	1.31

algorithms for the transformation of arbitrary linear computations to a maximally fast implementation for a fixed latency, and arbitrarily fast implementation have been developed. The proposed techniques produced for all used benchmark examples obtained results which are superior to the best previously published results.

REFERENCES

[1] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addision-Wesley, 1977.

[2] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, "Hyper-LP: Design system for power minimization using architectural transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12–31, Jan 1995.

[3] D. Chanoux, "A method of digital filter synthesis," masters thesis, Massachusetts Inst. Technol., Cambridge, MA, May 1969.

[4] J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version*. New York: Courant Institute of Mathematical Science, 1970.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[6] G. B. Danzig, W. Blattner, and M. R. Rao, "Finding a cycle in a graph with minimum cost to time ratio with application to as hip routing problem," in *Theory of Graphs*, P. Rosenstiehl, Ed. New York: Dunod, Paris and Gordon and Breach, 1967, pp. 77–84.

[7] M. D. Ercegovac and T. Lang, "Fast arithmetic for recursive computations," in *VLSI Signal Processing V* New York, 1992, pp. 14–28.

[8] A. Fettweiss, "Digital filter structures related to classical filter network," *Archiv Electronic Ubertragungstechnic*, vol. 25, pp. 79–89, 1971.

[9] G. Fettweiss and L. Thiele, "Algebriac recurrence transformations for massive parallelism," in *VLSI Signal Processing*, K. Yao, R. Jain, and W. Przytula, Eds. Piscataway, NJ: IEEE Press, 1992, pp. 332–341.

[10] B. Gold and K. L. Jordan, "A note on digital filter synthesis," *Proc. IEEE*, pp. 1717–1718, 1968.

[11] G. Goossens, J. Wandewalle, and H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems," presented at the 26th Design Automation Conf., Las Vegas, NV, 1989.

[12] I. Hong, D. Kirovski, and M. Potkonjak, "Potential-driven statistical ordering of transformations," in *34th ACM/IEEE DAC Design Automation Conf.*, Anaheim, CA, June 1997, pp. 347–352.

[13] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker, "Critical path minimization using retiming and algebraic speed-up," in *Proc. Design Automation Conf.*, 1993, pp. 573–577.

[14] J. M. Janssen, F. Catthoor, and H. De Man, "A specification invariant technique for operation cost minimization in flow-graphs," in *Proc. 7th Int. Symp. High-Level Synthesis*, Niagara-on-the-Lake, Canada, May 1994, pp. 146–151.

[15] J. M. Janssen, F. Catthoor, and H. De Man, "A specification invariant technique for regularity improvement between flow-graph clusters," in *European Design Automation Conf.*, Paris, France, 1996, pp. 138–143.

[16] R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computation for Uniform Recurrence Equation," *J. ACM*, vol. 14, no. 3, pp. 563–590, 1967.

[17] H. T. Kung, "New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences," *J. ACM*, vol. 23, no. 2, pp. 252–261, 1976.

[18] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C–36, pp. 24–35, Jan. 1987.

[19] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[20] H.-D. Lin and D. Messerschmitt, "Finite state machine has unlimited concurrency," *IEEE Trans. Circuits Syst.*, vol. 38, pp. 465–475, May 1991.

[21] D. A. Lobo and B. M. Pangrle, "Redundant operation creation: A scheduling optimization techniques," in *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 775–778.

[22] Y. Ofman, "On the algorithmic complexity of discrete functions," *Soviet Physics Doklady*, vol. 7, no. 7, pp. 589–591, 1963.

[23] A. V. Oppenheim and R. W. Shafer, *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[24] K. K. Parhi and D. Messerschmitt, "Pipeline interleaving and parallelism in recursive filters, Parts 1 and 2," *IEEE Trans. Acoust., Speech, Signal Processing*, pp. 1099–1117 and 1117–1134, 1989.

[25] ——, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. Comput.*, vol. 40, pp. 178–195, Feb. 1991.

[26] K. Parhi, "Pipelining in algorithms with quantizer loops," *IEEE Trans. Circuits Syst.*, vol. 38, pp. 745–754, July 1991.

[27] M. Potkonjak, "Algorithms for high-level synthesis resource utilization based approach," Ph.D. dissertsation, University of California at Berkeley, 1991.

[28] M. Potkonjak and J. Rabaey, "On unlimited parallelism of DSP arithmetic computations," in *Proc. 1993 Int. Conf. on Acoustic, Speech and Signal Processing*, 1993, pp. I.381–I.384.

[29] ——, "Optimizing resource utilization using transformations," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 277–292, Mar. 1994.

[30] ——, "Optimizing throughput and resource utilization pipelining transformations based approach," *J. VLSI Signal Processing*, vol. 8, no. 2, pp. 117–130, Oct. 1994.

[31] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architecture," *IEEE Design Test Comput.*, vol. 8, no. 2, pp. 40–51, June 1991.

[32] B. R. Rau, C. D. Glasser, and R. L. Pickard, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," in *Proc. 9th Int. Symp.Computer Architecture*, 1982, pp. 131–134.

[33] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM*, vol. 17, no. 4, pp. 715–728, 1970.

[34] M. B. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput," *IEEE Trans. VLSI Syst.*, vol. 3, no. 1, pp. 2–19, 1995.

[35] ——, "Power optimization in programmable processors and ASIC implementations of linear systems: Transformation-based approach," in *ACM/IEEE Design Automation Conf.*, 1996, pp. 343–348.

[36] J. D. Ullman, *Computational Aspects of VLSI*. Rockville, MD: Computer Science, 1984.

[37] H. B. Voelcker and E. E. Hartquist, "Digital filtering via block recursion," *IEEE Trans. Audio Electroacoust.*, vol. 18, no. 2, pp. 169–176, June 1970.

[38] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Programming Languages*, vol. 13, no. 2, pp. 181–210, 1991.

Miodrag Potkonjak received the EECS Ph.D. degree from University of California, Berkeley, in 1991.

He is an Associate Professor in the Computer Science Department, University of California, Los Angeles. He has published more than 150 papers in leading CAD and testing journals and conferences. He holds fivepatents. His research interests include intellectual property protection, system design and embedded systems.

Dr. Potkonjak received the Okawa Foundation Award, the NSF CAREER award, and a number of best paper awards.

Jan M. Rabaey received the EE and Ph.D degrees in applied sciences from the Katholieke Universiteit Leuven, Belgium, in 1978 and 1983, respectively.

He is a Professor in the Electrical Engineering and Computer Science departement of the University of California, Berkeley. He authored more than 150 papers in the area of signal processing and design automation. His current research interests include the exploration and synthesis of architectures and algorithms for digital signal processing systems and their interaction. He is, furthermore, active in various aspects of portable, distributed multimedia systems, including low-power design, networking, and design automation.

Dr. Rabaey received numerous scientific awards, including the 1985 IEEE Transactions on Computer Aided Design Best Paper Award (Circuits and Systems Society), the 1989 Presidential Young Investigator award, and the 1994 Signal Processing Society Senior Award.