Maximally Fast and Arbitrarily Fast Hardware Efficient Implementation of Linear and Feedback Linear Computations[1]

Miodrag Potkonjak
Dept. of Computer Science, University of California, Los Angeles, CA 90095

Jan M. Rabaey
Dept. of EECS, University of California, Berkeley, CA 94720

Keywords: digital signal processing, VLSI design, transformations, high level synthesis, linear computations, throughput, Horner Scheme

Abstract

By establishing a relationship between the basic properties of linear computations (additivity and homogeneity) and eight optimizing transformations (distributivity, associativity, commutativity, inverse and zero element law, common subexpression replication and elimination and constant propagation), a CAD 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 retiming and loop unrolling to the transformations set. During this process, a novel Horner pipelining scheme is used so that the 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. The new class includes popular nonlinear polynomial Volterra filters and widely used LMS and RLS adaptive filters. The effectiveness and low hardware overhead of the proposed techniques is illustrated on a number of academic and industrial designs.

1. Preliminary versions of this work appeared in the *Proc. IEEE/ACM 1992 ICCAD International Conference on Computer Aided Design*, November 1992 and in *Proc. 1993 IEEE ICASSP International Conference on Acoustic, Speech, and Signal Processing*, April 1993.

1.0 Introduction

1.1 Motivation and Goals

Throughput is widely recognized as the single, most important parameter of state-of-the art designs [Mit93]. Iteration bound, control and data dependences impose fundamental limits on achievable performance Transformations are universally accepted as the most efficient way in overcoming these limitations [Fis88, Kun88, Rab91]. However, until now the application of transformations for throughput enhancement was restricted to instances of specific computations [Hoc65, Sch84, Lu85, Par89, Fet90].

This paper addresses throughput optimization using transformations and has three major goals. The first goal is to demonstrate how for several important classes of computations the maximally fast implementation 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. The targeted classes include linear computations, polynomial filters and many types of adaptive filters. Note that this class of computation heavily dominates today DSP market [Opp89, Ahm90, Mit93]. The final goal is to build a basis and to give an impetus to the study of the efficient and effective ordering of transformations by studying the underlying synthesis mechanisms behind the proposed procedures. We also discuss how the new approach can be used for increasingly important design parameter - power.

1.2 Paper Road Map

In the next two sections we survey preliminaries and the related work. The new approach is informally introduced in Section 4. A rigorous and detailed presentation of the algorithms to transform a linear computation into maximally fast implementation and arbitrarily fast implementation and the underlying theoretical results are presented in Section 5. In Section 6 the results and algorithms are generalized so that nonlinear computations can be efficiently transformed for throughput. Experimental results on benchmark examples are shown in Section 7. We conclude the paper by restating all theoretical and algorithmic results from a transformation point of view.

2.0 Preliminaries

2.1 Linear and Feedback Linear Computations

Linear systems are a standard topic in textbooks on control theory and digital signal processing [e.g.Kai80, Opp89]. A particularly important subclass of linear systems is the one with the time-invariant property. Linearity and time-invariance can be combined into a mathematical framework, which facilitates analyzis and synthesis. But most importantly, linear computations have "the widest signal processing application range" [Opp89]. For the sake of completeness, we briefly summarize the basic facts about linear systems.

Definition:

A linear system is a system L such that:

(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).

These two properties are often compacted in the following symbolic definition of linearity:

$$L\{a_1 x_1 + a_2 x_2\} = a_1 L\{x_1\} + a_2 L\{x_2\},$$

for any a_1 and a_2 from the set of real or complex numbers. The property expressed by the last formula is most often called the principle of superposition, and can be straightforwardly generalized to a superposition of many inputs [Opp89].

Linear computations use as computational elements only addition, subtraction and multiplication with constants. Due to the superposition property, 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 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 [Wae50]. So all results presented here are directly applicable to computations on groups where, for example, addition corresponds to max and multiplication to addition [Fet90], or addition corresponds to logical or and multiplication to logical and.

Classification of computation into only two classes, linear and nonlinear, is for many purposes a coarse one. We will introduce a new class of computations, feedback linear, in this subsection. 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 as possible a class of computation enable provably maximally fast and arbitrarily fast implementations.

Informally and 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. Besides linear computations, a significant part of nonlinear computations belong to this class. Note that those computation may have variable-variable multiplications and divisions between primary inputs or their linear combinations of primary inputs themselves, or between primary inputs and algorithmic delays in feedback loops. Even multiplication and divisions between the algorithmic delays are allowed, as long as they do not belong to feedback loops.

More formally, feedback linear computations can be defined as those who can be described using the following set of equations.

$$X[n+1] = A*X[\mathrm{n}] + B*U[n]$$

$$Y[n] = C*X[\mathrm{n}] + D*U[n]$$

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 which has as entries functions which do not depend on feedback states X[n]. There is no restriction of functions which forms the entries of matrices B, C and D.

The importance of feedback linear computations is emphasized by the fact that many of modern VLSI applications belong to this class. While, as we already noted, linear computations are currently most widely used in DSP, control, and telecommunications application, the rest of the feedback linear computation class is probably the fastest growing segment of those markets [Ahm90]. The feedback linear class of computations includes Kalman, LMS adaptive, block LMS, direct RLS adaptive, Cholesky RLS adaptive, QR-RLS adaptive, and nonlinear Volterra polynomial filters [Mit93]. It is estimated that a great majority of 16 and 24 bit fixed-point DSP processors is used today in adaptive LMS filters in cellular and portable phones [Ahm90]. The adaptive filters are also regularly used in other high volume applications, such as data and voice cancelers, for equalization of magnetic disk channels, and for equalization of digital wideband packet radio networks [Mit93]. Volterra filters have an important role in modeling biological phenomena, echo cancelation, and image and video applications [Mit93].

Identification of feedback linear computation is again, as in the of linear computations, simple and can be done rapidly in polynomial, worst case quadratic, time. The first step is the identification of cycles using the standard depth first search algorithm for labeling of strongly connected components [Cor90]. All strongly connected components with more than one node, belong to cycles. All other nodes do not belong to cycles. The second step is a linear time checking 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 and software can be used for constant propagation [Weg91, Rab91].

2.2 Assumptions and Definitions

We assume that an arbitrary linear computation is given. In the rest of the paper we interchangeably use terms linear computation and arbitrary linear computation. The later phrase is used when we want to stress that we do not impose any restrictions on considered linear computation. The assumed computational model is homogeneous synchronous data flow [Lee87]. This model is widely used in many application domains (DSP, video and image processing, communications, control, information theory applications) and is also the facto standard model for individual tasks in hard real-time research and development. There is an available set of execution units (e.g. adder, multiplier) so that each arithmetic or logic operation requires a certain fixed time. On the computation a set of transformations can be applied. The considered set of transformations encompasses distributivity, associativity, commutativity, inverse element law, zero element law, common subexpression elimination, common subexpression replication, and constant propagation.

Four more definitions are used in the rest of the paper:

Maximally fast implementation of an algorithm (or 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 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 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 $A_2 * T_2^2 < A_1 * T_1^2$. (In several cases we prove even more strict definition which requires $A_2 * T_2 < A_1 * T_1$) [Ull84].

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

3.0 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 as an effective approach for the improvement of the implementation of computations in a number of research and development domains, including compilers, VLSI design, computer arithmetic, mathematics, theoretical computer science, numerical computations, logic synthesis, high level synthesis and algorithm design.

Despite the fact that the importance of transformations was realized early in computer science and engineering, the difficulty of applying them properly and their high optimization potential still makes them a prime research topic. There exist excellent references on the use of transformations in compilers [Wul75, Aho77, Fis88]. Therefore, we will primarily concentrate our discussion of previous work on the use of transformations in CAD, VLSI design and in particular high level synthesis. Also, we will restrict our attention to optimization issues. For discussion of formal properties of transformations and other transformations issues, see [McF93] and [Wal89]. Furthermore, since more than a hundred primitive transformations have been tabulated [Aho77, Ban93, Bac94, Fis88, Par95, Wul75], we will mainly discuss those which have a dominant role with respect to our major target, linear computations. For the formal definition of all transformations discussed here, we refer to the standard compiler reference works [Aho77, Fis88].

Common subexpression elimination is regularly used in many compilers and discussed in great detail in the compiler literature [Fis88, Aho77]. Most often the treatment of common subexpression in compiler research and development is based on value numbering [Coc70] and Sethi-Ullman numbering [Set70] techniques in peephole optimization framework [McK65]. Common subexpression replication has only recently been getting attention. Jouppi [Jou88] made an insightful observation that for parallel code executed on either super-pipelined or super-scalar architectures, most of today's optimizing compilers will actually reduce the amount of instruction-level parallelism due to the extensive use of transformations such as common subexpression elimination. Unfortunately, he did not pursue this direction any further.

The situation is similar for high level synthesis systems. While the use of common subexpression elimination is widely advocated and practiced in several systems [Rab91], it seems that the only attempt of applying common subexpression replication can be attributed to Lobo and Pangrle [Lob91]. Although this effort was limited to a very specific instance of computation (a part of the 5th order wave digital elliptic filter) and was not combined with other transformations, they succeeded to reduce the critical path from initial 17 (or 16 control cycles after the application of the Leiserson-Saxe retiming algorithm) to 12 cycles. It is interesting to note that the transformation proposed by Lobo and Pangrle has been widely used in a much more general setting under name of Skipp-Add in the theoretical computer science community [Mil88]. Skipp-Add is essentially a combination of two transformations, common subexpression replication and associativity, where the first one acts as an enabling transformation for the second one, which actually reduces the critical path.

While constant propagation can be solved straightforwardly when the computation does not involve conditionals and loops, it can be easily shown that the constant propagation problem is undecidable in general [Weg91]. The standard procedure to address this problem is to use so called 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. An excellent survey of the most popular constant propagation algorithms, as well as their generalization, can be found in [Weg91].

Associativity, distributivity and commutativity are the three most often used algebraic transformations [Wae50]. Most often they are treated under the paradigm of tree-height reduction, although several authors discuss in detail the critical path minimization of general DAG's (directed acyclic graph) [Tri87, Har89, Lob91, Dun92]. 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 [Pot91].

Other algebraic transformations are related to other axioms in algebraic structures, e.g. inverse and zero element law [Wul75, Wae50], 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 [Pot91]. Iqbal et al. [Iqb93] used the zero inverse element law as part of algebraic speed-up procedure which transforms 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 [Par88b, McN90] i. As noticed by several authors, there is a very close relationship between retiming and pipelining [Gla85, Pot92][1]. 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 in polynomial time [Lei91].When the goal is minimal area or power, the problem has been proven to be NP-complete [Pot91]. There is one more important relationship between two transformations: pipelining and software pipelining [Rau82, Goo89]. As described in [Pot92], 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. Therefore only minor modifications are needed to modify optimization techniques for one to another.

Loop unfolding (loop unrolling, loop unwinding, look-ahead computations) is one of those transformations which is regularly used in almost all optimizing compilers and many high level synthesis systems [Rab91]. Recently, an interesting result was obtained: when only retiming and loop unfolding are considered and the goal is to minimize the critical path, then the order in which transformations are applied does not influence the final result [Cha92].

At each step of the optimization process, a compiler or a high level synthesis system must select a transformation which is both legal and desirable. By far the most popular and widely used approach for this task is the already mentioned peephole optimization [McK65], where the compiler considers only a very limited section of code in order to apply one by one iteratively and locally all available transformations. Another popular approach, in particular in logic synthesis [Bra84], is a static ordering where the order of transformations is given a priori, most often in the form of a script.

1. When we refer to pipelining we mean functional pipelining.

Neural networks and other probabilistic techniques often provide a good trade-off between the run time and the quality of solution when no clear insight in a more targeted optimization process is available. Recently, neural network techniques were proposed for a probabilistic approach to the application of transformations [Fox89]. Compound approaches which combine probabilistic (probabilistic sampling and simulated annealing) and local heuristic approaches were used to demonstrate a significant reductions in area [Pot94] and power [Cha93].

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 $x_i = a_i + x_{i-1}$. The linear recurrences problem has been studied in mathematics for centuries [Wae85]. 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 Y. Ofman to design a fast parallel circuit for the addition of binary numbers. 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 [Hwa79]. Ercegovac and Lang recently surveyed this line of research [Erc92].

In 1965, Hockney proposed a fast direct solution of Poisson's equation based on exploring parallelism in solving linear recurrences. The flurry of research in numerical computation along those lines resulted in several algorithms for of a number of widely used matrix and finite difference computations in applied numerical analysis [Buz70, Sto73]. Two excellent reviews of the application of fast methods for linear recurrences in numerical analysis and computations are survey articles by Temperton [Tem82] and Swarztrauber [Swa77].

In 1967, a study of linear recurrence in the theoretical computer science community was initiated by Karp, Miller, and Winograd. Ladner and Fisher [Lad80] and Brent and Kung [Bre80] showed efficient general purpose circuits and an efficient VLSI layout for a scan circuit which computes linear recurrences, and named the problem parallel prefix computation Kung [Kun76] 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 [Gol68]. Soon it was applied on a variety of signal processing problems [Cha69, Vol70]. 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 IIR filters and several classes of adaptive filters [Sch84, Mes88, Fet90, Lin91]. In particular, we point out work done by Parhi and Messerschmitt which does not just summarize many of previous techniques and reduce their hardware overhead, but also introduces a number of new application domains [Par89].

Since the early seventies the treatment of linear recurrences has been widely studied in the compiler community, mainly when compilation to vector and highly pipelined computers are targeted [Che75, Gaj81].

3.1 What is new?

It is instructive and important to compare the new technique with the related research. The maximally fast implementation of linear computation (Section 5) is a widely applicable topic which has not been studied previously. The proposed arbitrarily fast solution maintains the AT product constant (Section 6, Theorem 5) and hence is more hardware efficient than any of the previously proposed techniques. This is achieved using a novel Horner scheme based pipelining technique (see Section 5.2) 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 (see Section 8) provides several advantages. It enables that all results are directly applicable in several domains. For instance, the methods presented here can be directly applied on optimization of time loops in DSP arithmetic computations as well as control loops in software compilers. Also, for the first time it provides a framework where all known approaches are treated as special cases of the general approach. Moreover, it provides for the first time an arbitrarily fast implementation for a number of increasingly important new computational classes, such as nonlinear polynomial filters (Section 6). It also indicates that the class of feedback linear computation covers all computations where arbitrary speed-up is achievable (Section 6). Finally, the CAD approach to the problem provides a convenient way that an arbitrary linear or feedback linear computation is transformed in maximally and arbitrarily fast forms automatically. Until now state of art techniques required a significant creative and development effort for this task, and the resulting structures were regularly reported in journal publications [Gol68, Vol70, Sch84, Mes88, Par89, Fet90]. 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 [Sri96].

4.0 New Approach: Idea and Introductory Examples

Probably the best way to introduce a basic idea is to use simple, yet non-trivial examples. Figure 1 shows the control-data flow graph (CDFG) of a 3-port serial adaptor which is regularly used in many ladder digital filter structures, such as elliptic wave filters [Fet71, Hon84]. 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 Figure 1 and equals 7 clock cycles, assuming that each operation takes one cycle.

Iteration bound [Dan67, Fet76, Rei81] imposes a fundamental limit on the highest throughput which can be achieved using an arbitrary level of pipelining and arbitrary retiming. Therefore when a

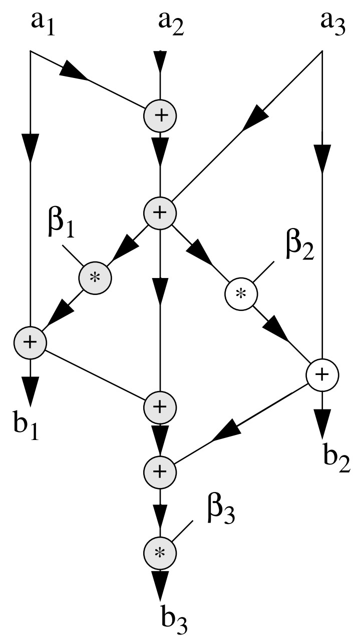


FIGURE 1. CDFG of a 3-port serial adapter: shaded nodes belong to critical path

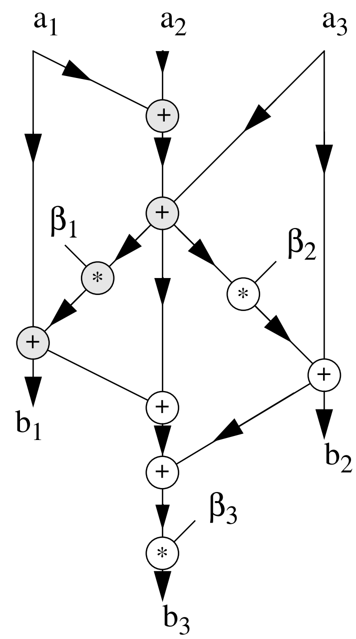


FIGURE 2. CDFG of a 3-port serial adapter: shaded nodes denote the set of nodes on which output b_1 depends

given computation is a part of a feedback loop, pipelining and retiming are ineffective as critical path minimization techniques. Applying algebraic transformations (such as associativity and distributivity) 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 to extend the overall search space. This can be achieved in the following way.

Figure 2 shows the same adaptor, however now the nodes which are in transitive fan-in of the output $b1$ are shaded. Obviously, in order to correctly compute this output it is sufficient to take into account only this part of the computation. Figure 3 shows the functional dependences between all output nodes b_i and input nodes a_i. To get from the structure of Figure 1 to the result of Figure 3, we have effectively replicated all subexpressions common between the different output variables and hence made the computational trees independent.The resulting CDFG has only additions and multiplications with constants as computational nodes. So, each output is a linear combination (weighted sum) of the inputs. This observation leads directly to a significant reduction in the length of critical path. Algebraic laws, and in particular associativity, 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 dependences of Figure 4 are obtained. The CDFG of the restructured computation is presented in Figure 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 3 clock cycles. Although the speed-up is signifi-

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

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

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

FIGURE 3. Functional dependences between output and input nodes for the serial adaptor exampl

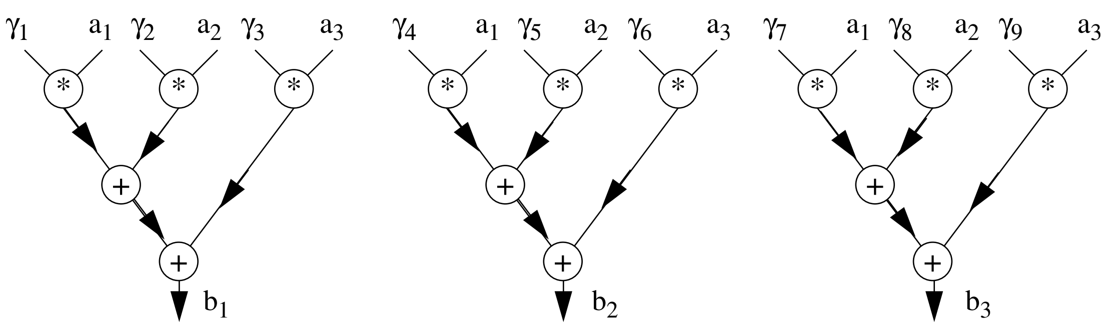
$$b_1 = (\beta_1 + 1) * a_1 + \beta_1 * a_2 + \beta_1 * a_3$$

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

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

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

cant, it can be achieved in a simple and effective way. 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



$\gamma_1 = \beta_1 + 1$; $\gamma_2 = \beta_1$; $\gamma_3 = \beta_1$; $\gamma_4 = \beta_2$; $\gamma_5 = \beta_2$; $\gamma_6 = \beta_2 + 1$; $\gamma_7 = \beta_3 * (\beta_1 + \beta_2 + 2)$; $\gamma_8 = \beta_3 * (\beta_1 + \beta_2 + 2)$; $\gamma_9 = \beta_3 * (\beta_1 + \beta_2 + 1)$;

FIGURE 5. CDFG for restructured 3-port serial adapter. Note that some of coefficients are 1, for those multiplications are not needed

reduce the required number of multiplications by one by applying distributivity $(\gamma_4 * a_1 + \gamma_5 * a_2 = \gamma_4 * (a_1 + a_2))$. 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. This is even true when the designs have feedback loops which prevent the use of pipelining. The ideas behind this concept had, until now, only been demonstrated manually for restricted application areas[e.g. Par89, Lin91]. Even for those cases, the new procedure will trade hardware for throughput more efficiently.

First Iteration:

$$
\begin{aligned}
x_1 &= In_1 + a^*x_0 + b^*y_0 \\
y_1 &= x_0 \\
Out_1 &= x_1 + c^*x_0 + d^*y_0
\end{aligned}
\qquad \text{(a)}
$$

Second Iteration:

$$
\begin{aligned}
x_2 &= In_2 + a^*x_1 + b^*y_1 \\
y_2 &= x_1 \\
Out_2 &= x_2 + c^*x_1 + d^*y_1
\end{aligned}
\qquad \text{(b)}
$$

After Unrolling and Algebraic Transformations:

$$
\begin{aligned}
x_2 &= In_2 + a^*In_1 + (a^*a + b)\,x_0 + a^*b^*y_0 \\
y_2 &= In_1 + a^*x_0 + b^*y_0 \\
Out_1 &= In_1 + (a + c)\,x_0 + (b + d)\,y_0 \\
Out_2 &= In_2 + (a + c)\,In_1 + (a^*a + b + c^*a + d)\,x_0 + (a^*b + c^*b)\,y_0
\end{aligned}
\qquad \text{(c)}
$$

FIGURE 6. Functional dependences for 2nd order IIR filter: (a) and (b) first two iteration before and after unfolding

To convey the ideas behind the procedure, we will again use a small but real life example, as shown in Figure 7. This figure shows the popular second order IIR filter direct form II design, the standard building block in many recursive filters [Opp89]. Notice that the filter has two feedback loops, which prevent the direct usage of pipelining or (time) loop unfolding. Figure 6a 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 into analytical expression of Figure 6b. The CDFG of the unfolded IIR filter is shown in Figure 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 Figure 6c. Notice that the graph of Figure 6c has exactly the same input/output relationship as the initial specification. The

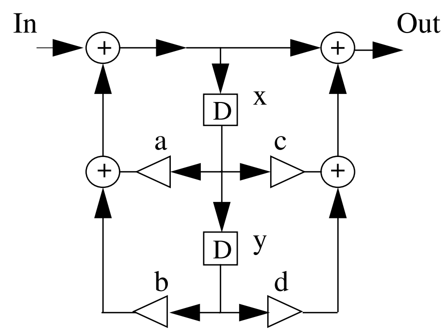


FIGURE 7. Second order IIR filter

minimal execution time for this graph is only 3 clock cycles, which is identical to the critical path of the original graph of Figure 6a. As a result, the throughput has been actually doubled as two iterations can be performed in the time initially needed for one. Figure 9 shows the computational structure of the unfolded implementation for the number of iterations.

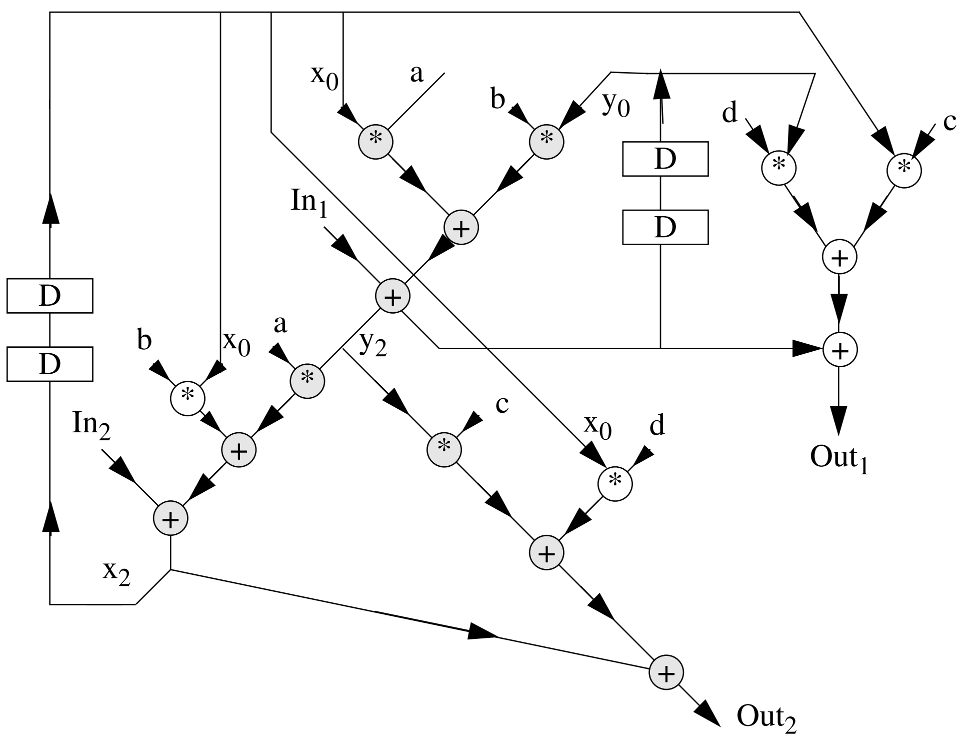
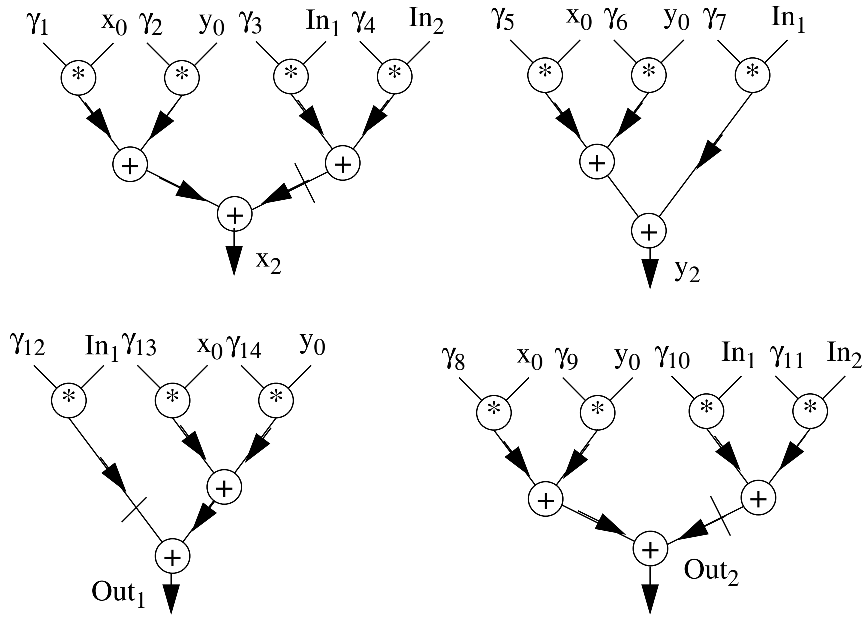


FIGURE 8. CDFG of the unfolded 2nd order direct form-II IIR filter

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 3 input variables. After organizing all those computations so that all multiplications are done in the first cycle,



γ_1=a*a + b; γ_3=a; γ_5=a; γ_7=1; γ_9=a*b + c*b; γ_{11}=1; γ_{11}=a+c;
γ_2=a*b; γ_4=1; γ_6=b; γ_8=a*a + b + c*a + d; γ_{10}=a + c; γ_{12}=1; γ_{12}=b+d;

FIGURE 9. Computational Structure of the Unfolded and restructured Implementation of the 2nd order IIR filter. Crossed edges denote positions of pipelining delays.

followed by additions organized in a tree format, the critical path will stay unaltered, while we will process more and more iterations. If the basic iteration *is unfolded N* times, the number of simultaneously processed samples and throughput will increase also 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 targeted. The techniques presented in this paper are used as a basis for answering this questions, as presented in [Sri95]. Srivastava and Potkonjak [Sri95] presented a technique which provides a provably optimal combination of latency and throughput in a sense than any other feasible implementation of a given linear computation has either larger latency or smaller throughput. The techniques presented in this paper still have two important advantageous even when both latency and throughput are targeted during design: they are computationally more efficient, they result in more efficient implementations and they can be generalized to wide class of computations.

5.0 Optimizing Linear System for Throughput: Algorithms and Theoretical Results

Due to dramatic improvements in silicon technology, design for minimal area has become less stringent in recent years and trading off area for performance has become a viable alternative for getting market advantage. Even when the goal by itself is not sheer performance, it is often advantageous to have an option for improving the speed of design. For example, the fastest growing segment of the electronics market currently is the area of portable communications and computing equipment. Portability is mainly constrained by power requirements. Since there are several approaches which effectively trade throughput for power reduction [Cha92], much of the improvement in speed can be traded for power reduction if the effective switching capacitance is not increased or increased at lower rate.

Although the primary goal of the proposed algorithms is to maximize throughput, several other 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 three algorithms for:

(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.

5.1 Maximally Fast Implementation of Linear Computations

In Section 2, we 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 CDFG 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. Even when the number of nodes in the graph is polynomial, the number of paths between the input nodes and an output node can be exponential. One such example is shown in Figure 10. For such a structure with a critical path N, the number of the paths from node I to node O equals $\binom{N-1}{\frac{N-1}{2}}$. While the number of nodes in the DCFG equals N^2, the number of paths is larger then $((N-1)/2)!$, which can be approximated for large N using the Stirling formula $\sqrt{\pi N}\left(\frac{N}{2e}\right)^{N/2}$.

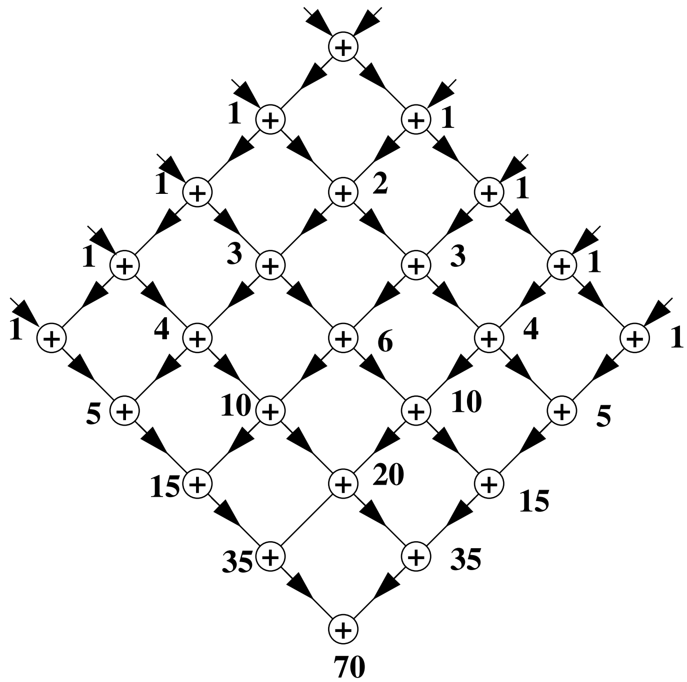


FIGURE 10. The example of the DCFG structure where the number of nodes grows polynomially, but the number of dependences (paths between the input and the output node) grows exponentially

Still, the relationship between an output node and the input nodes can be obtained in polynomial time using dynamic programming as presented in [Pol83]. 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 CDFG. 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 1 is assigned to this input node and the value 0 to all other inputs nodes. The numbers next to the CDFG nodes in Figure 10. demonstrate this approach, assuming that all nodes are additions. As a result of these steps, the multiplication weights for all inputs are obtained. This procedure, assigning 1 to one input node and 0 to the rest, is repeated for each input node. As the end result, we have the exact weight coefficient of how each output node (either delay or primary output) depend on each input (either delay or primary input). It is interesting to note that during this step both zero and inverse element laws are implicitly applied.

The second step, pipelining of primary inputs, is an enabling transformation which greatly reduces the influence of primary inputs. If the CDFG has only one primary input and all outputs have a weight 1 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 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 first two steps act as an enabler for this procedure. The last step does not influence the length of the critical path. It is used for the optimization of the secondary goal, area. This step is explained in subsection 4.3.

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 as a means to eliminate influence of inputs, is the fastest possible. The factor N, used in the formulation, denotes the number of delays in feedback cycles.

Theorem 1:

Given a computational graph of a 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 $\lceil \log N + 1 \rceil$ ***computational levels [2]. 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 the factor 2. We need one more level for multiplications by constants. The proposed algorithm reaches this lower bound. Q.E.D.

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 can be used to minimize the maximum number of algorithmic delay inputs on which an output depends. Currently, we use the probabilistic sampling algorithm [Pot91] for this complex task. The only difference is that the objective function is altered to reflect a new optimization goal.

5.2 Algorithm for Arbitrarily Fast Implementation of Linear Computations

When the introduction of additional latency into the design is allowed, we can additionally reduce the critical path by trading latency and hardware for throughput. In this subsection, we will show how the algorithm presented in the previous subsection, can be used as a starting point and one of the main building blocks for developing an efficient technique for the arbitrary speedup of linear computation.

2. All the logarithms in this paper are base 2.

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 CDFG dependences can be expressed in the following form:

$$X[n+1] = A*X[\mathrm{n}] + B*U[n]@1$$

$$Y[n] = C*X[\mathrm{n}] + D*U[n]@1$$

Sign @ denotes the introduction of pipeline stages. All four matrices A, B, C, and D have as entries only constants. Matrix A has as entries weight coefficients between delays, matrix B has as entries weight coefficients which describe how delays depend on primary inputs. Similarly, matrices C and D have as entries coefficients which indicate how the primary outputs depend on the delays and the primary inputs, respectively. This form is very similar to the canonical state space representation of a linear system [Kai80]. The only difference is that the maximally fast procedure assumes that all primary inputs in both the first and the second equations and that the part of CDFG for computing primary outputs, outside of cycles, are 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. Figure 11 shows the first four iterations of the targeted linear computation. It also shows n-th iteration of the maximally fast implementation of the targeted linear computation

$$X_2 = AX_1 + BU_1$$
$$Y_1 = CX_1 + DU_1$$

$$X_3 = AX_2 + BU_2$$
$$Y_2 = CX_2 + DU_2$$

$$X_4 = AX_3 + BU_3$$
$$Y_3 = CX_3 + DU_3$$

$$X_5 = AX_4 + BU_4$$
$$Y_4 = CX_4 + DU_4$$

$$X_{n+1} = AX_n + BU_n$$
$$Y_n = CX_n + DU_n$$

FIGURE 11. First four and n-th iteration of a linear computation.

Figure 12 shows the same set of equations, now after substitution so that each left side is expressed only in terms of the initial delays and primary inputs.

$$X_2 = AX_1 + BU_1$$
$$Y_1 = CX_1 + DU_1$$

$$X_3 = A^2X_1 + ABU_1 + BU_2$$
$$Y_2 = CAX_1 + CBU_1 + DU_2$$

$$X_4 = A^3X_1 + A^2BU + ABU_2 + BU_3$$
$$Y_3 = CA^2X_1 + CABU_1 + CBU_2 + DU_3$$

$$X_5 = A^4X_1 + A^3BU_1 + A^2BU_2 + ABU_3 + BU_4$$
$$Y_4 = CA^3X_1 + CA^2BU_1 + CABU_2 + CBU_3 + DU_4$$

$$X_{n+1} = A^nX_1 + A^{n-1}BU_1 + A^{n-2}BU_2 + \ldots + ABU_{n-1} + BU_n$$
$$Y_n = CA^{n-1}X_1 + CA^{n-2}BU_1 + CA^{n-3}BU_2 + \ldots + CBU_{n-1} + DU_n$$

FIGURE 12. First four and the n-th iteration of linear computation, after all variables on the left side of equations are expressed in terms of primary inputs and delays.

In order to correctly compute the semi-infinite stream of data we need to compute the variables shown on the left side of the linear dependences shown in Figure 13. Note that we not only need to that we compute all 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 shown in Figure 13 directly indicates that regardless of the how many iterations we implement simultaneously, the

$$X_{n+1} = A^nX_1 + A^{n-1}BU_1 + A^{n-2}BU_2 + \ldots + ABU_{n-1} + BU_n$$
$$Y_1 = CX_1 + DU_1$$
$$Y_2 = CAX_1 + CBU_1 + DU_2$$
$$Y_3 = CA^2X_1 + CABU_1 + CBU_2 + DU_3$$
$$\vdots$$
$$Y_n = CA^{n-1}X_1 + CA^{n-2}BU_1 + CA^{n-2}BU_2 + \ldots + CBU_{n-1} + DU_n$$

FIGURE 13. Arbitrarily Fast Implementation of Linear Computation: Necessary and Sufficient Set of Computations.

critical path of the unfolded and transformed structure is not unaltered.

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 DCFG 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 CDFG 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 one 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, considers both retiming and algebraic transformations simultaneously, but is associated with a more difficult optimization problem. For this task, we provide a probabilistic heuristic algorithm.

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 ([Orl88] as suggested by Leiserson and Saxe [Lei91] (where |V| is the number of operational elements, |E| is the number of edges in the CDFG). 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 [Pot91].

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). The identification of all delays on which a particular output depends can be easily done in polynomial time using the depth-first search starting from all outputs [Cor90]. However, restructuring of the targeted CDFG 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. Note that a variety of probabilistic algorithms for the second alternative can be readily derived. Also note that both preprocessing alternatives are optional. Even when they are not used at all, the algorithm will always result in the arbitrarily fast implementation. The importance of preprocessing alternatives is limited to cases where not just high throughput is requested, but where there is also a need for low latency.

The second step, unfolding of the initial iteration over 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 [Ull84, page 48]. 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.

$$
\begin{aligned}
Y_1 &= CX_1 + DU_1 \\
Y_2 &= CAX_1 + C(BU_1) + DU_2 \\
Y_2 &= CA^2X_1 + C(A(BU_1) + BU_2) + DU_3 \\
Y_4 &= CA^3X_1 + C(A(A(BU_1) + BU_2) + BU_3) + DU_4 \\
&\vdots \\
Y_n &= CA^{n-1}X_1 + C(A(\qquad)) + DU_{n-1}
\end{aligned}
$$

FIGURE 14. Application of generalized Horner scheme on the arbitrarily fast implementation of an arbitrary linear computation.

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

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

to the following form:

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

Therefore, an arbitrary polynomial can be computed using at most n additions and n multiplications. Horner proposed this rule in 1819, and although it was used by Isaac Newton a century and half earlier, his name is associated with this widely used transformation. The excellent exposure of Horner's rule and the number of its generalization are presented by Knuth [Knu81].

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 shown in Figure 13, so that this overhead is reduced to linear increase. The resulting structure is shown in Figure 14 using the functional dependency form. The part of the pipelined computation which depends only on the primary inputs is shown graphically in Figure 15. 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. Q.E.D.

An interesting observation is that if the level of unfolding is small, when the number of delays in feedback loops is large and the number of inputs and outputs is small in several initial unfoldings, the direct computation using functional dependences shown in Figure 13 may be the more economical alternative. However, when the number of unfoldings is large, the new scheme is superior to the initial, regardless of the number of inputs.

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.

5.3 Minimization of the Number of Operations in the Fast Implementation of Linear Computation

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 four transformations, common subexpression elimi-

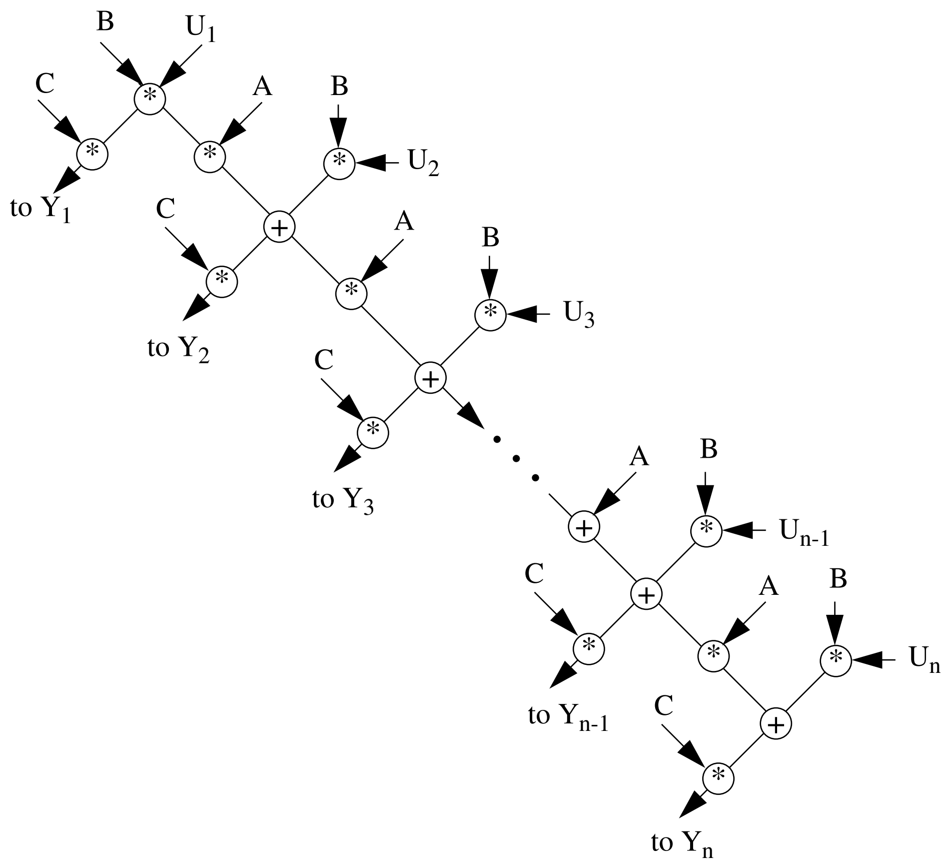


FIGURE 15. The part of linear CDFG on which generalized Horner scheme is applied. The structure can be pipelined to an arbitrary level.

nation, 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 a

	i1	**i2**	**i3**	**i4**	**i5**	**i6**	**i7**	**i8**
o1	1	0	0	0	0	0	0	0
o2	126	125	112	56	0	0	56	56
o3	160	160	152	80	9	0	80	80
o4	7	7	7	6	0	0	7	7
o5	140	140	133	70	8	0	70	70
o6	144	144	144	144	0	9	232	240
o7	162	162	162	162	0	10	261	270
o8	150	150	150	150	0	0	250	269
o9	135	135	135	135	0	0	225	243

TABLE 1. Functional dependences between inputs and outputs for 5th order elliptical filter

high throughput, but also small area. An additional benefit is that by reducing the number of operations, we also reduce the number of switching events. Therefore, the algorithms presented in this subsection significantly enhance the application domain of maximally fast and arbitrarily fast techniques and makes them an attractive option for producing area and power competitive designs.

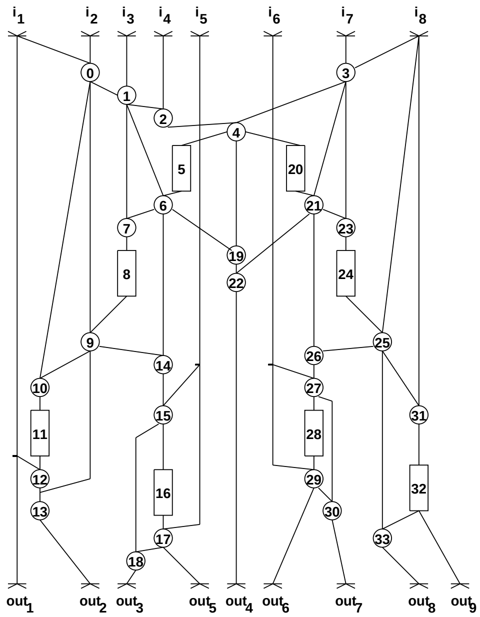


FIGURE 16. Fifth Order Elliptical Wave Digital Filter. Rectangles denote two cycle long multiplications circles denote unit cycle additions.

We will use a 5th order wave digital elliptical filter as an illustrative example to introduce both the problem formulation and the algorithm for solving this new optimization task. Figure 11 shows the computational graph of this popular benchmark [Dew85]. There are 8 inputs for one iteration of the filter (7 delays and one primary input i_1) and 8 outputs (seven delays and one primary output o_9).

The entries in Table 1 show the functional dependences between outputs and inputs (assuming that constant weighting factors for multiplications 5, 20, 8, 24, 11, 16, 28 and 32 (see Figure 11) are set to the values of 2, 3, 4, 5, 6, 7, 8, 9, respectively [3]). Table 1 and Figure 16 show for each output O_i the inputs I_i on which it depends. By counting those dependences, 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.

Figure 17 shows the same functional dependences 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, o_8) are the same and have to be computed only once by invoking common subexpression elimination.

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

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 first and primarily

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

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

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

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

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

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

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

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

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

FIGURE 17. Functional dependences of the 5th order elliptical wave digital filter

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.

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 $\mathbf{o_7}$ the application of distributivity, augmented with commutativity, ($\mathbf{o_5} = 140 * i_1 + 140 * i_2 + 133 * i_3 + 70 * i_4 + 8 * i_5 + 70 * i_7 + 70 * i_8 = 140 * (i_1 + i_2) + 133 * i_3 + 70 * (i_4 + i_7 + i_8) + 8 * i_5$) reduces the number of multiplications by 4.

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. It is mainly so because we intentionally selected all constants which are used for multiplication to be different, in order to demonstrate that even then the number of operations in the fast implementations can be reduced significantly.

The third step is the application of reduction in strength [Aho77]. Reduction in strength is a popular compiler transformation which substitutes a more expensive operation with a less expensive. Most often multiplications are replaced by additions. Until now, its application range was mainly restricted to array index calculation during loop optimization in software compilers. Here we propose its new use: replacement of multiplications by addition in maximally and arbitrarily fast implementations of linear computations. 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 5th order filter (i_5 in $\mathbf{o_3}$ and $\mathbf{o_5}$; i_6 in $\mathbf{o_6}$ and $\mathbf{o_7}$, and i_8 in $\mathbf{o_7}$ and $\mathbf{o_8}$). We trade one multiplication for one addition by replacing computations $int_1 = c * i_i$ and $int_2 = (c+1)\ i_i$ by $int1 = c * i_i$ and $int_2 = int_1 + 1$.

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 tarde multiplication for additions. One instance of this trade-off is $\mathbf{o_2}$, i_1 and i_2.

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 $\mathbf{o_6}$, $\mathbf{o_7}$, and $\mathbf{o_8}$. It is, of course, sufficient to calculate it only once. Also note that during the computation of $\mathbf{o_2}$, $\mathbf{o_3}$, and $\mathbf{o_5}$ it is advantageous to form an intermediate result $i_4 + i_7 + i_8$, while when $\mathbf{o_6}$, $\mathbf{o_7}$, and $\mathbf{o_8}$ 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 [Pot93].

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 in Fast implementations of Linear Computations:

(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 the MCM iterative matching algorithm [Pot93].

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.

Figure 18 shows the final CDFG for efficient calculation of the maximally fast 5th order elliptical wave digital filter.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FIGURE 18. The CDFG of the 5th order elliptical wave digital filter after the application of the maximally fast procedure and the procedure for the minimization of the number of operations

For the 5th order elliptical filter example, only 23 multiplications and 29 additions are finally needed for a critical path of 5 cycles (assuming that a multiplication takes 2 cycles). The original implementation uses 8 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 4 multipliers and 3 adders, which is 3 adders less than in the previously best published implementation [Lob91] 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 [Pot92]. 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. Note that the application of functional pipelining has to be carefully conducted, since overpipelining introduces a high need for registers [Pot92]. Therefore, we position pipelining algorithmic delays just in front of multiplication, and achieve a good trade-off for simultaneous minimization of both execution units and registers.

6.0 Optimizing Feedback Linear Computations for Throughput: Algorithms and Theoretical Results

Although linear computations are exceptionally widely-used in many applications, a natural, interesting and important question is to exactly characterize the power of the approach and algorithms presented in the previous section in order to establish which is the widest class of computations where the algorithms are effective. In [Pot92] we presented the piecewise linear approach which generalizes the previous version of the algorithms presented in Section 4 and enables the application of basic methodology to arbitrary nonlinear computations. Although the experimental result presented there clearly demonstrated the high effectiveness of that approach, the optimality of maximally and arbitrarily fast techniques were lost. Here, we will take a different direction from the beginning; the primary goal is to develop a set of sufficient and necessary conditions when maximally and arbitrarily fast implementation is feasible. Even more importantly, we will present a constructive technique for this task, i.e. for all cases when the provably maximal and arbitrary throughput are achievable, we will provide a transformation procedure which leads to such performance.

In particular, in this section we will present how the fast throughput techniques from the previous section can be applied, with very minor modifications, so that any computation which belongs to a wide and important class of feedback state linear computations can be transformed so that its throughput is maximally increased.

The modifications are based on two simple, but far reaching observations and one well-know result from theoretical computer science. The first 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 can not be handled easily. When the target is throughput, the only possible bottlenecks are feedback cycles. Recall that all operations out of feedback cycles can be always readily pipelined to any requested level. This observation is used to remove any constraints for arbitrarily and maximally fast computations on the computational structure outside of feedback cycles.

The second key observation is that the only limitations (bottlenecks) to maximally fast procedure are the size of the considered CDFG and requested linearity. The theoretical science result which we will use is established by Kung [Kun76]. He proved that if a recurrence is nonlinear it cannot be arbitrarily sped-up.

It is always easy to remove the first limitation imposed by the second observation by doing unfolding along a time loop. The combination of the first observation and the second limitation of the second observation indicates that linearity in the feedback cycles is sufficient to achieve arbitrary speed-up. Finally, since the recurrence is a special case of feedback system, the combination of Kung's theoretical result and the second observation indicates that the feedback linear systems are the largest class of computation for which arbitrary speed-up is achievable.

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 can be applied in general case. Therefore, each additional application of operator A, induces in the worst case quadratic overhead.

Still, there is one point which is worth noticing briefly. It is that a translation from an arbitrary form of the targeted feedback linear computation to the following computational structure which is the starting point starting for the application of maximally fast and arbitrarily fast implementation, can be easily accomplished by using common subexpression replication.

$$X[n+1] = A*X[\mathrm{n}] + B*U[n]$$

$$Y[n] = C*X[\mathrm{n}] + D*U[n]$$

It is important to stress again that only algorithmic delays which are in feedback loops are relevant and limiting factor for throughput in both linear and feedback linear computations. All other delays which do not belong to come cycle in the initial specifications do not impose constraints on throughput performances. Figure 19 shows a 2nd order Volterra filter. Only two delays close to the output (denoted by y@1 and y@2) are constraining throughput; two delays close to the input (x@1 and x@2) are in feed-forward parts and do not influence maximum achievable throughput.

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 addi-

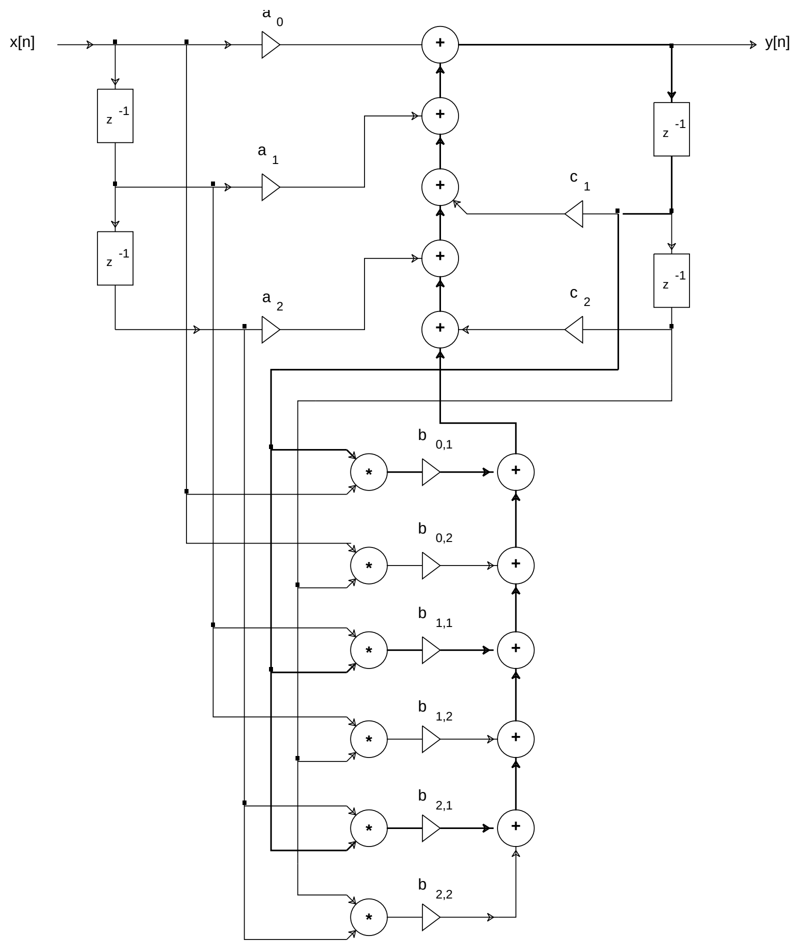


FIGURE 19. Second order fixed coefficients Volterra polynomial filter. Example of feedback linear computation. Only delays y@1, and y@2 belong to feedback cycles.

tions, subtractions, and constant multiplications in feedback cycles. A more formal definition is that constant feedback computations are computations which can be represented in form:

$$X[n+1] = A^*X[\mathrm{n}] + B^*U[n]$$

$$Y[n] = C^*X[\mathrm{n}] + D^*U[n]$$

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 logN + 1 computational levels and is provided by the algorithm presented in Section 4.1.

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.

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. We conclude this section by restating key new theoretical results. Maximally/arbitrarily fast implementation always exist for a (arbitrary) linear or feedback linear computation. The proposed procedures are constructive and result in one of many optimal solutions. The maximally/arbitrarily forms produced by the algorithms presented in the paper are, in general case, not unique. A number of transformations, such as commutativity and common subexpression elimination, can be applied on them without alternating the throughput of the implementation. The key property of arbitrary fast implementations is that they preserve AT and AT^2 products for linear and feedback linear computations respectively.

7.0 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,

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 2: Fast Implementation of Linear ProgramsICP - initial critical path**; FCP -** critical path after the application of the algorithm for fast implementation of linear programs (FLP); **IFCP -** ratio of critical path before and after application of FLP**; IA -** initial area; **FA -** Final area**; FIA -** ratio of the final and the initial and area; **IFAT -** ratio of the initial and the final AT product. The description of examples is provided in Section 6.

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 3: Fast Implementation of Feedback Linear Programs: ICP - initial critical path**; FCP -** critical path after the application of the algorithm for fast implementation of linear programs (FLP)**, IFCP -** ratio of critical path before and after application of FLP**, IA -** initial area; **FA -** Final area**; FIA -** ratio of the final and the initial and area; **IFAT -** ratio of the initial and the final AT product. The description of examples is provided in Section 6**.**

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 2. 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 - 5th order elliptical wave digital filter, 7IIR- 7th order IIR cascade filter, 8IIR - 8th order Avenhaus filter, 11 - 11th order IIR Chebishev filter, 12IIR - 12th order Butterworth filter, 18IIR - 18th order IIIR filter, 22IIR - 22nd order IIR filter, and DAC- NEC digital-to-analog converter. The average and median improvement in throughput are by fac-

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

Table 4: Application of Arbitrarily FAst Procedure on 5th order linear controller: CP - effective critical path for one iteration; **Energy -** 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 called voltage per sample respectively. The columns represent initial design and designs after application of n (n=1,...,10) times unfolded arbitrarily fast procedure.

tors 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.

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

Table 5: Application of Arbitrarily Fast Procedure on Volterra Filter: CP - effective critical path for one iteration; **Energy -** energy at 5 V per sample; **Energy -** energy at called voltage per sample; **CPR, AreaR, EnergyR, EnergySR -** ratio between initial and final critical path, area, energy at 5 V, and energy at the called voltage per sample respectively. The columns represent initial design and designs after application of n (n=1,2,3) times unfolded arbitrarily fast procedure

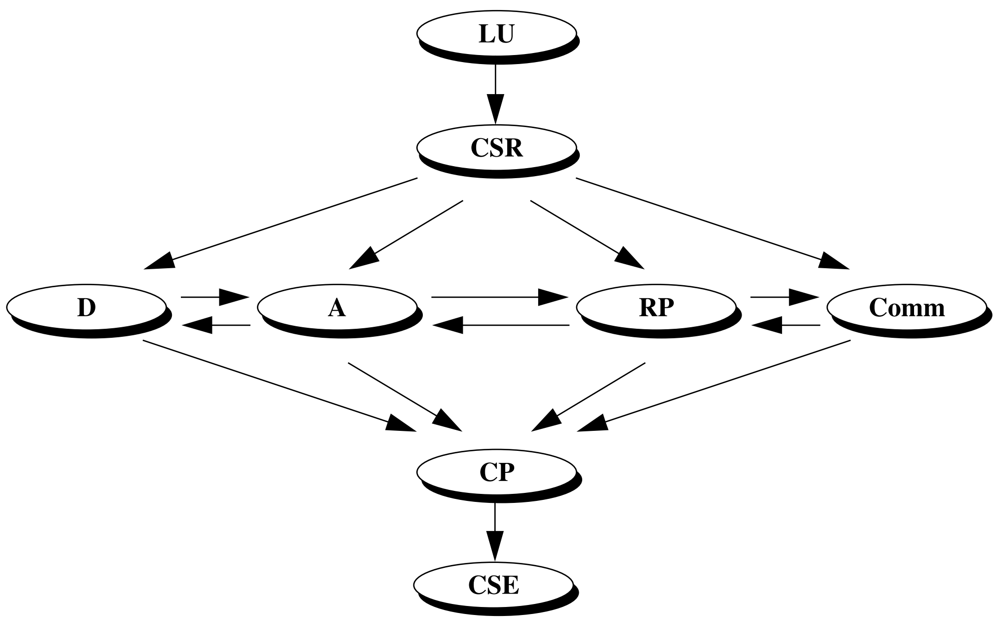
Two examples, lin4 and 4th order LMS adaptive filter were studied for analysis of arbitrarily fast algorithm. The behavior of the two examples was drastically different. While for the linear computation example, overhead in both power and area were modest, the feedback linear example experienced very high area overhead. For example, when the throughput of the 5th 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 [Cha92], then Table 4 indicates that power can be reduced by a factor of almost 5 .

However, for the feedback linear example the area overhead was more than 12 times when throughput is increased 6 times. Similarly, the effective capacitance increased at an even higher pace. Interestingly, still it was advantageous to apply voltage calling, and trade improved throughput for energy saving. For example, when two unfoldings are applied in the arbitrarily fast procedure, the energy was reduced to almost half of its original value. However, additional speed-up was counterproductive as a way to save energy, since capacitance overhead was exceptionally high.

8.0 Linear Computations and Transformation Ordering

Although it is possible to present all ideas and results of this paper without explicitly referring to transformations, it is very important and instructional to treat the whole problem as the task of applying transformations. This approach will not only provide a better insight into the effectiveness (and so importance) of particular transformations, but also a clearer view on the problem of transformation ordering.

It was realized from the early applications in software, logic synthesis and high level synthesis compilers that ordering of transformations is the key factor in exploring their full potential. However, while in all three areas impressive results are achieved in applying specific transformations (for example loop transformation in software compilers [Aho77, Fis85], very little work was done on transformation ordering. The new viewpoint gives strong hints why and how some transformations can act as enabling transformations (which temporarily increase the objective function, but can trigger other transformations which result in more substantial improvement). Finally, the results about both effectiveness and ordering are general and independent of the particular application domains, so that they can be used not just in high level synthesis but also in logic synthesis and compilers. Also, this explanation provides the validation proof of the correctness of the proposed approach.



LU - Loop Unfolding

CSR - Common Subexpression Replication

CP - Constant Propagation

CSE - Common Subexpression Elimination

A - Associativity

D - Distributivity

RP - Retiming (and Pipelining)

Comm - Commutativity

FIGURE 20. The enabling relationship between transformation used in this paper.

The first step in the algorithm for a maximally fast implementation and fixed latency is a decomposition of the computations for each output. This step is nothing else but a common subexpression replication applied on all places in a CDFG where we have broadcasts. This step does not effect the critical path and has as a negative side effect a large increase in the number of operations. The motivation behind this step is to enable the application of associativity, which was inhibited by common subexpressions.

During the second step, we concentrate all constants by which a particular input is multiplied with this input using algebraic transformations. So now, by using constant propagation, we can reduce both the number of operations in the CDFG and the critical path.

The last step is the application of associativity so that chains of additions in the DCFG are transformed into a binary tree. The number of operations is not altered, but the critical path is reduced from a linear size (as a function of the number of addition) to a logarithmic size.

Therefore, the main point in the procedure can be summarized in the following way: Common subexpression elimination, which was initially applied by the original designer, prevents the application of other transformations. This constraint can be removed by common subexpression replication which reveals opportunity for the application of both algebraic transformations and constant propagation.

If we analyze the procedure for the arbitrarily fast implementation of linear computations using a similar procedure, the following conclusion can be drawn. The first two steps, loop unfolding and common subexpression replication, are enabling transformations for the next steps, where algebraic laws and constant propagation are applied to reduce the critical path.

These observations are summarized in Figure 14. Essentially, the ordering between several transformations is determined by which transformation enables higher effectiveness for another one. So, we can conclude that loop unrolling should be applied first, followed by common subexpression replication because both are enabling transformations for distributivity, associativity, retiming and commutativity. Although both loop unfolding and common subexpression are enabling transformations, loop unfolding should be applied first since it enables the application of common subexpression replication to a larger extent.

The second interesting point is that the relationship and therefore ordering between two transformations can be altered by their relationship with a third one. For example, although the ordering of loop unrolling and retiming is unimportant for the critical path reduction [Cha92], since loop unrolling enables associativity (through the replication of common subexpression) and associativity and retiming enable each other, associativity and retiming [Pot91] should be applied simultaneously for maximum effectiveness.

The most important future work resulting from this research is in building a relationship among other commonly used transformations, such that the transformational search space can be restricted without hampering their effectiveness. It is important to notice that all observations made here regarding the ordering of transformations are not restricted to the domain of linear or feedback linear computations.

Furthermore, all those observations are correct, regardless of the goal of the transformations (optimization of speed, area and power). However, the type of transformations applied and their effectiveness are both goal and domain dependent.

9.0 Conclusion

Using the relationship between the superposition property for linear computations and the set of optimizing transformations (distributivity, associativity, common subexpression replication and elimination, constant propagation, retiming and loop unfolding), algorithms for the transformation of arbitrary linear computations to a maximally fast implementation for a fixed latency, and arbitrarily fast implementations implementation have been developed. The proposed techniques produced for all used benchmark examples obtained results which are superior to the best published previously. For example, the critical path of the 5th order wave digital elliptical filter was reduced from the initial 17 cycles to only 5 cycles without introducing any additional latency (this compared to the previous best of 12 cycles). The same example can be arbitrarily sped-up using the arbitrary while the AT product is maintained. We also discussed how some of the obtained insights can be used for the efficient ordering of transformations in high level synthesis systems.

10.0 References

[Ahm90] H.M. Ahmed: "Directions in DSP Processors", IEEE Journal on Selected Area in Communications, Vol. 8, No. 8, pp. 1420-1427, 1990.

[Aho77] A.V. Aho, J.D. Ullman: "Principles of Compiler Design", Addison-Wesley, Reading, MA, 1977.

[Bac94] D.F. Bacon, S.L. Graham, O.J. Sharp, "Compiler Transformations for High Performance Computing", ACM Computing Surveys, Vol. 26, No. 4, pp. 345-420.

[Ban93] U. Banerjee, R. Eigenmann, A. Nicolau, D.A. Padua, "Automatic Program Parallelization", Proc. of IEEE, Vol. 81, No. 2, pp. 211-243, 1993.

[Bra84] R.K. Brayton, G.D. Hachtel, C.T. MvMullen, A.L. Sangiovanni-Vincentelli: "Logic Minimization Algorithms for VLSI Synthesis", Kluwer, Boston, MA, 1984.

[Buz70] B. Buzbee, G. Golub, C. Nielson: "On direct methods for solving Poisson's equation", SIAM Journal on Numerical analysis, Vol. 7, No. 4, pp. 627-656, 1970.

[Cha92] A.P. Chandrakasan, M. Potkonjak, J.Rabaey, R.W. Brodersen: "Hyper-LP: Design System for Power Minimization using Architectural Transformations", IEEE ICCAD-92, pp. 300-303, 1992.

[Cha69] D. Chanoux: "A method of Digital Filter Synthesis", M.S. thesis, MIT, Cambridge, MA, May 1969.

[Che75] S-C. Chen, D.J. Kuck: "Time and Parallel Processor Bounds for Linear Recurrence Systems", IEEE Trans. on Computers, Vol. 24, No. 7, pp. 701-717.

[Coc70] J. Cocke, J.T. Schwartz: Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version, New York: Courant Institute of Mathematical Science, 1970.

[Cor90] T.H. Cormen, C.E. Leiserson, R.L. Rivest: "Introduction to Algorithms", The MIT Press, Cambridge, MA, 1990.

[Dan67] G.B. Danzig, W. Blattner, M.R. Rao: "Finding a cycle in a graph with minimum cost to time ratio with application to as hip routing problem", *Theory of Graphs*, (P. Rosenstiehl, Ed.), pp. 77-84, New York, NY:Dunod, Paris and Gordon and Breach, 1967.

[Dew85] P. Dewilde, E. Depretter, R. Nouta: "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms", in "VLSI and Modern Signal processing", ed. by S.Y. Kung, H.J. Whitehouse, T. Kailath, pp. 257-264, Prentice Hall, Englewood Cliffs, N.J., 1985

[Dun92] P. Duncan, et al.: "Hi-Pass: A Computer Aided Synthesis System for Fully Parallel Digital Signal Processing ASIC's", IEEE ICASSP-92, pp. V-605-608, San Francisco, 1992.

[Erc92] M.D. Ercegovac, T. Lang: "Fast Arithmetic for Recursive Computations", VLSI Signal Processing V, pp. 14-28, IEEE Press, New York, NY, 1992.

[Fet71] A. Fettweis: "Digital Filter Structures Related to Classical Filter Network" in "Selected Papers in Digital Signal Processing, II", edited by DSP Committee IEEE ASSP Society, pp. 475-485, originally published in Archiv Electronic Ubertragungstechnic, Vol. 25, pp. 79-89, 1971.

[Fet76] A. Fettweis: "Realizability of Digital Filter Networks", Arch. Elek. Ubertragung, Vol. 30, No. 2, pp, 90-96, 1976.

[Fet90] A. Fettweis, H. Meyr, L. Thiele: "Algorithm Transformations for Unlimited Parallelism", *IEEE International Symposium on Circuits and Systems*, pp. 1756-1759, New Orleans, 1990.

[Fis88] C.N. Fischer, R.J. Le Blank: "Crafting a Compiler", The Benjamin/Cummings Publishing Co., Menlo Park, CA, 1988

[Fox89] G.C. Fox, J.G. Koller: "Code Generation by a Generalized Neural Network", Journal of Parallel and Distributed Computing, Vol. 7, No. 2, pp. 388-410, 1989.

[Gaj81] D.D. Gajski: "An Algorithm for Solving Linear Recurrences Systems on Parallel and Pipelined Machines", Vol. 30, No. 3, pp. 190-206, 1981.

[Gla85] L. Glasser, D. Dobberpuhl: The Design and Analysis of a VLSI Circuits", Addison-Wesley, Reading, MA, 1985.

[Gol68] B. Gold, K.L. Jordan: "A Note on Digital Filter Synthesis", Proc. of IEEE, pp. 1717-1718, 1968.

[Goo89] G. Goossens, J. Wandewalle, H. De Man: "Loop optimization in register-transfer scheduling for DSP-systems", *26th Design Automation Conference*, pp. 826-831, Las Vegas, NV, 1989.

[Har89] R. Hartley, A. Casavant: "Tree-height Minimization in Pipelined Architectures", IEEE ICCAD, pp.112-115, 1989.

[Hoc65] R. Hockney: "A fast direct solution of Poissons's equation using Fourier analysis", Journal of the ACM, Vol. 12, No. 1, pp. 95-113, 1965.

[Hon84] M. Honig, D. Messerschmitt: "Adaptive filters: structures, algorithms, and applications", Kluwer, Boston, MA, 1984.

[Hwa79] K. Hwang: "Computer Arithmetic: Principles, Architecture, and Design", John Wiley, New York, NY, 1979.

[Iqb93] Z. Iqbal, M. Potkonjak, S. Dey, A. Parker: "Critical path Minimization using retiming and algebraic speed-up, pp. 573-577, ACM/IEEE Design Automation Conference, 1993.

[Jou88] N.P. Jouppi: "Superscalar vs. superpipelined machines", Comp. Architectures News, Vol. 16., No. 3, pp. 71-80, 1988.

[Kai80] T. Kailath: "Linear systems". Prentice-Hall, Englewood Cliffs, N.J., 1980.

[Kar67] R.M. Karp, R.E. Miller, S. Winograd: "The Organization of Computation for Uniform Recurrence Equation", Journal of the ACM, Vol. 14, No. 3, pp. 563-590, 1967.

[[Kil73] G.A. Kildall: "A unified approach to global program optimization", Conference Recordings of the First ACM Symp. on Principles of Programming Languages", pp. 194-206.

[Knu81] D.E. Knuth: "The Art of Computer Programming: Volume 2 / Seminumarical Algorithms", 2nd edition, Reading, MA, 1981.

[Kog73] P.Koge, H. Stone: "A parallel algorithm for the efficient solution of a general class of recurrence equations", IEEE Trans. of Comp., Aug 1973.

[Kun76] H.T. Kung: "New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences", Journal of the ACM, Vol. 23, No. 2, pp. 252-261, 1976.

[Lad80] R.E. Ladner, M.J. Fisher: "Parallel Prefix Computation", Journal of the ACM, Vol. 27, No. 4, pp. 831-838, 1980.

[Lee87] E.A. Lee and D.G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing.” *IEEE Transactions on Computers*, Vol. 36, No. 1, pp. 24–35, 1987.

[Lei91] C.E. Leiserson, J.B. Saxe: “Retiming Synchronous Circuitry”, Algorithmica, Vol. 6, pp. 5-35, 1991

[Lin91] H.-D. Lin, D.Messerschmitt: “Finite State Machine has Unlimited Concurrency”, IEEE Trans. on CAS, Vol. 38, No. 5, pp. 465-475, 1991.

[Lob91] D.A. Lobo, B.M. Pangrle: “Redundant Operation Creation: A Scheduling Optimization Technique”, 28th ACM/IEEE Design Automation Conference, pp. 775-778.

[[McF93] M. C. McFarland: “Formal Analysis of Correctness of Behavioral Transformations”, Formal Methods in System Design, Vol. 2, No. 3, pp. 231-257.

[McK65] W.M. McKeeman: “Peephole optimization”, Communications of the ACM, Vol. 8, No. 7, pp. 443-444, 1965.

[McN90] K. McNall, A. Casavant: “Automatic Operator Configuration in the Synthesis of Pipelined Architectures”, 27th ACM/IEEE DAC, Orlando, FL, pp. 174-179., 1990.

[Mes88] D.G. Messerschmitt: “Breaking the recursive bottleneck”, in “Performance Limits in Communication Theory and Practice”, J.K. Skwirzinsky, Ed., Kluwer, Amsterdam, 1988.

[Mil88] G.L.Miller, V. Ramachandran, E. Kaltofen: “Efficient Parallel Evaluation of Straight-Line Code and Arithmetic Circuits,”, *SIAM Journal on Computing*, Vol. 17, No 4, pp. 687-695, 1988.

[Mit93] S. K. Mitra, J.F. Kaiser: “Handbook for Digital Signal Processing”, John Wiley, New York, NY, 1993.

[Off63] Y. Ofman: “On the Algorithmic Complexity of Discrete Functions”, Soviet Physics Doklady, Vol. 7, No. 7, pp. 589-591, 1963.

[Opp89] A.V. Oppenheim, R.W. Shafer: “Discrete-time Signal Processing”, Prentice Hall, Englewood Cliffs, NJ, 1989.

[Orl88] J.B. Orlin: “A faster strongly polynomial minimum cost flow algorithm”, Proceedings of the 20th ACM Symposium on Theory of Computing, ACM, pp. 377-387, 1988.

[Par88b] N. Park, A.C. Parker: “Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications”, IEEE Transaction on CAD for IC, Vol 7, No. 3, pp. 356-370, 1988.

[Par89] K. K. Parhi, D. Messerschmitt: “Pipeline interleaving and parallelism in recursive Filters, Parts 1 and 2”, IEEE Trans. on ASSP, Vol. 37, pp. 1099-1117 and 1117-1134, 1989.

[Par95] K.K. Parhi, “High-Level Algorithm and Architecture Transformations for DSP Synthesis”, Journal of VLSI Signal Processing, Vol. 9, No. 1-2, pp. 121-143, 1995.

[Pol83] G. Polya, R.E. Tarjan, D.R. Woods: “Notes on introductory combinatorics”, Birkhauser, Boston, MA 1983.

[Pot91] M. Potkonjak: “Algorithms for High Level Synthesis Resource Utilization Based Approach”, *Ph.D. Thesis*, University of California at Berkeley, 1991.

Pot92b] M. Potkonjak, J. Rabaey: “Pipelining: Just Another Transformation”, International Conference on Application Specific Processors, pp. 163-177, IEEE Computer Society Press, Los Alamitos, CA, 1992.

[Pot92a] M. Potkonjak, J. Rabaey: “Maximally Fast and Arbitrarily Fast Implementation of Linear Computations”, ICCAD-92, pp. 304-308, 1992.

[Pot93] M. Potkonjak, J. Rabaey: “On Unlimited Parallelism of DSP Arithmetic Computations”, 1993 International Conference on Acoustic, Speech and Signal Processing, pp. I.381-I.384, 1993.

[Rab91] J. Rabaey, et al.: “Fast Prototyping of Data Path Intensive Architecture”, IEEE Design and Test, pp.40-51, 1991.

[Rau82] B.R. Rau, C.D. Glasser, R.L. Pickard: “Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support”, *Proc. 9th Intl. Symposium on Computer Architecture*, pp. 131-134, 1982.

[Ren81] M. Renfors, Y. Neuvo: “The Maximum Sampling Rate of Digital Filters under Hardware Speed Constraints”, IEEE Trans. on Circuits and Systems, Vol. 26, No. 3, pp. 196-202, 1982.

[Sch84] D.A. Schwartz, T.P. Barnwell: “Increasing the Parallelism of Filters Through Transformation to Block State Variable Form”, International Conference on Acoustic, Speech, and Signal Processing, pp. XXX 1984.

[Set70] R. Sethi, J.D. Ullman: “The generation of optimal code for arithmetic expressions”, Journal of the ACM, Vol. 17, No. 4, pp. 715-728, 1970.

[Sto73] H. Stone: “An efficient algorithm for the solution of a tridiagonal linear system of equations”, Journal of ACM, Vol. 20, No. 1, pp. 27-38, 1973.

[Sri95] M. B. Srivastava, M. Potkonjak, “Optimum and Heuristic Transformation Techniques for Simultaneous Optimization of Latency and Throughput”, IEEE Transaction on Very Large Scale Integration (VLSI) Systems, Vol. 3, No. 1, pp. 2-19, 1995.

[Sri96] M. B. Srivastava, M. Potkonjak, “Power Optimization in Programmable Processors and ASIC Implementations of Linear Systems: Transformation-based Approach”, 1996 ACM/IEEE Design aUtomation Conference, to be published.

[Swa77] P. Swarztrauber: “The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson’s equation on a rectangle”, SIAM Review, Vol. 19, No. 4, pp. 490-501.

[Tem79] C. Temperton: “Direct methods for the solution of the discrete Poisson equation: some comparisons”, Journal on Computational Physics, Vol. 31, No. 1, pp. 1-20., 1982.

[Tri87] H. Trickey: “Flamel: A High-Level Hardware Compiler”, IEEE Transaction on CAD, Vol. 6, No. 2, pp. 259-269, 1987.

[Ull84] J.D. Ullman: “Computational Aspects of VLSI”, Computer Science Press, Rockville, MD, 1984.

[Voe70] H.B. Voelcker, E.E. Hartquist: “Digital Filtering via Block Recursion”, IEEE Trans. on Audio Electroacoustics, Vol. 18. No. 2, pp. 169-176.

[Wal89] R.A. Walker, D.E. Thomas: “Behavioral transformations for algorithmic level IC design”, IEEE Trans. on CAD, Vol. 8, No. 10., pp. 1115-1128, 1991.

[Wae50]. B. L. van der Waerden: “Modern algebra”, Ungar, New York, 1950-1953.

[Wae85] B. L. van der Waerden: “A History of Algebra: From al-Khwarizmi to Emmy Noether”, Springer-Verlag, Berlin, Germany, 1985.

[Weg91] M.N. Wegman, F. K. Zadeck: “Constant Propagation with Conditional Branches”, ACM Transactions on Programming Languages, Vol. 13, No. 2, pp. 181-210, 1991.

[Wul75] W.Wulf, R.K. Johnson, C.B. Weinstock, S.O. Hobbs, C.M. Geschke: The Design of Optimizing Compiler”, North Holland, New York, NY, 1975.