Maximizing the Throughput of High Performance DSP Applications Using Behavioral Transformations

Shan-Hsi Huang Jan M. Rabaey

Department of EECS, University of California
Berkeley, California 94720, USA

Abstract

Meeting the stringent throughput requirements of high performance DSP applications is a challenging task. Extensive optimization of the computational structure is essential to satisfy these constraints. This paper proposes a new transformational approach for performance optimization. This approach consists of an ordered set of transformations, including algebraic transformations, loop unrolling, and retiming/pipelining, aimed at speeding up both recursive and non-recursive, as well as linear and non-linear applications. Impressive and close to optimal speed-up's have been obtained for a large range of benchmark examples.

1 Introduction

Getting the highest possible throughput is often the goal in high performance DSP applications. The conventional approach to obtain high performance is to exploit parallelism through the use of pipelining. Unfortunately, for recursive applications (i.e. applications with feedback), the *iteration bound*[1] limits the speed-up that can be obtained by pipelining [12]. To speed up an application beyond the iteration bound, it is necessary to explore other transformations.

A framework containing a set of primitive transformations is established. The transformations we considered can be classified into three main categories: algebraic, temporal (retiming/pipelining) and loop transformations (loop unrolling). Since the ordering in which these transformations are applied has an important impact on the final result, the *enabling and postponing principle* is introduced (Sec 3). This principle is the major guideline for the application of transformations in the framework.

In order to reduce the iteration bound, moving operations out of cycles with algebraic laws is essential. Among algebraic transformations, *associativity* is the most important one that can *directly* move operations out of cycles to reduce the bound. New algorithms to apply associativity and algebraic transformations to minimize the iteration bound are proposed in Sec 4 and Sec 5, respectively. Together with the existing algorithms for loop unrolling[2] [14] and retiming/pipelining [11], an ordered transformational framework is provided in Sec 7. With the framework, a dataflow graph can be automatically transformed to satisfy the given throughput constraint at the cost of bounded overhead in area.

1. The iteration bound is defined as follows, where $T(C)$ is the total computation time of a cycle C and $D(C)$ is the number of (sample) delays in C.

$$IB = max_{\forall \text{cycles } C}\left(\frac{T(C)}{D(C)}\right)$$

2 Previous work

When several transformations are available, a common approach is to establish an interactive environment to allow a designer to manually explore implementation alternatives. Potkonjak [16] first brought up the idea of enabling transformations in high level synthesis. Certain transformations, which could achieve the attempted goal might not be applicable initially. However, by applying some other transformations first (called *enabling transformations*), they might eventually become activated. Using this idea, he proposed an approach to automatically maximize the throughput of a linear computation with loop unrolling, retiming, and algebraic laws. In addition to the constraints of linearity, his approach focuses on minimizing the critical path.

For applications with high throughput constraints, look-ahead computation has been intensively utilized. This technique can arbitrarily speed up linear recurrences[3] [9][6][12][13][2][3]. Parhi succeeded in systematically applying the look-ahead computation on linear time-invariant applications by using transfer functions or state space representations. The idea behind look-ahead computation is to remove operations from cycles such that the iteration bound is reduced. This idea is generalized in our approach.

3 Transformation ordering

In the presence of a library of transformations, a deterministic approach towards ordering these transformations is preferential when automating the optimization process. This ordering varies

2. The time loop unrolling refers to the well known loop unfolding or block processing.

3. Kung [10] proved that a non-linear polynomial recurrence can only be sped up by a constant factor no matter how many processors are available.

1066-1409/94 $3.00 © 1994 IEEE

with the objective. In this paper, we will address the ordering for maximizing the throughput.

Since several transformations, like associativity, retiming, and pipelining, are candidates for reducing the critical path, it is not clear which ones should be selected and in what order. If the iteration bound is within the timing constraints, applying pipelining/retiming is sufficient to reduce the critical path to satisfy the constraints. If not, *reducing the iteration bound* is the real target behind performance optimization[1].

Since loop unrolling only duplicates the loop bodies (including both computational operations and delay operators), the iteration bound cannot be changed by this transformation alone. Retiming, which moves delays around, cannot change the number of delays in a cycle. Pipelining, which inserts delays to reduce the critical path, has no capability of inserting delays into a cycle. Therefore, the achievable speed-up with these transformations is limited by the iteration bound. In order to reduce the bound, moving operations out of cycles using algebraic laws is essential.

Algebraic transformations consist of associativity, commutativity, (reverse) distributivity, common sub-expression replication/elimination (CSR/CSE), and constant folding[2]. Among these, associativity is the most important transformation that can *directly* move operations out of cycles to reduce the iteration bound. The dependency relations between associativity and other transformations can be obtained with the *enabling principle*.

Since constant folding is often enabled by associativity, and commutativity can enable associativity without extra cost, these three should be applied together. These transformations do not increase the number of operations in a structure and, therefore, their impact on the area is minimal. They are the *kernel* techniques in our framework. Distributivity and CSR, on the other hand, are labeled as enabling transformations. Distributivity swaps addition and multiplication operations to put the same type of operations together, thus enabling associativity. Furthermore, associativity and distributivity cannot be applied to operations with multiple fanouts. This inhibiting factor can be removed with CSR by creating redundant operations.

The applicability of algebraic transformations is related to the number of operations in the graph. The more operations in the graph, the more likely it is that algebraic transformations can be applied. Loop unrolling creates extra operations in cycles. This enables algebraic transformations. In the event that algebraic transformations cannot reduce the bound alone, loop unrolling is applied to enable them. Since at most a linear speed-up can be achieved with the help of time loop unrolling, the minimum unrolling factor can be determined.

When a feasible iteration bound is obtained with loop unrolling and algebraic transformations, retiming/pipelining may be used to reduce the critical path down to the bound. Finally, CSE is applied to minimize the area without affecting the obtained critical path. With the above analysis, a complete ordering of transformations, aimed at reducing the iteration bound, is obtained (Figure 1).

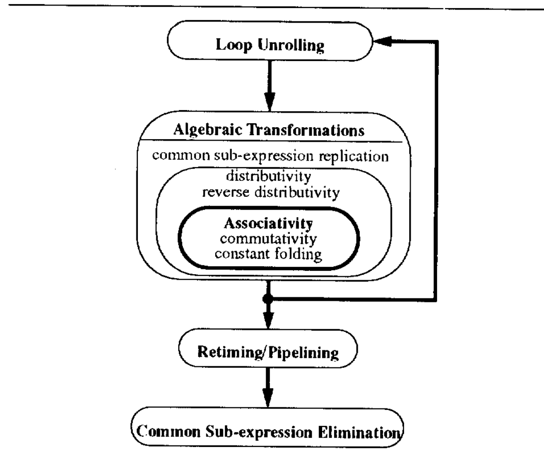

Figure 1: Dependency ordering of transformations

However, arbitrarily activating enablers, such as CSR or distributivity, may induce area overhead. This produces an optimized but impractical design. In practice, these enabling transformations should be postponed until they are really necessary. In other words, the *redundant* enablers should not be applied at all (postponed forever). This is the *postponing principle*. This principle can prevent the sacrificing of extra overhead in area. With the postponing principle, area is simultaneously and indirectly minimized while optimizing speed. The approach with only the enabling principle is to apply the transformations in a *top-down* fashion. The enablers are applied first. On the other hand, with the *enabling and postponing (EP) principle*, the transformations are applied in a *bottom-up* fashion. Associativity is applied first. The enablers are only applied when needed (ideally). The introduction of the postponing concept is one of the main contributions of this paper (compared to e.g. [16]).

Based on the *EP* principle, we propose an algorithmic methodology which builds up an ordered transformational framework. In the framework, a dataflow graph can be automatically transformed to meet timing constraints. The algorithm for loop unrolling by Parhi [14] and the one for retiming by Leiserson and Saxe[11] have been integrated into this framework. The retiming algorithm can be used for the pipelining transformation by inserting delays at the inputs [17]. To complete the framework, we propose an algorithm to apply the kernel transformations (associativity, etc.) in the next section. This algorithm is then augmented with other enabling algebraic transformations in Sec 5.

4 Kernel transformation - associativity

This section discusses the kernel transformations of the framework: associativity, commutativity, and constant folding. Hartley [5] proposed an efficient way of using associativity and commutativity to minimize the critical path. His approach, modi-

1. Note that the real bound is the smallest integer which is greater than the iteration bound and the maximal computation time of an operation. For simplicity, we assume the iteration bound as defined in the previous page. The proposed methodology can be easily extended to handle the general case.

2. Distributivity is defined as [a * (b+c) ⇒ a*b + a*c]. Reverse distributivity is defined as [a*b + a*c ⇒ a * (b+c)]. Constant folding is to precompute an operation of constant value, e.g. c = 1 + 2 is replaced by 3.

fied and augmented with constant folding, is used to minimize the iteration bound. The algorithm is briefly outlined as follows:

Step 1: collapse operation trees with associativity.
Step 2: separate operands in cycles from those not in cycles.
Step 3: split operations in cycles to minimize the iteration bound.
Step 4: split operations not in cycles to minimize the critical path.

Step 1 transforms a structure into the canonical form by collapsing an operation tree into a multiple-input operation. During collapsing, the commutativity law is implicitly applied. Subtraction (division) is treated in the same associative class as addition (multiplication) by negating (inverting) the appropriate operands. The negative (inverse) sign will be recovered during splitting (Step 3 &4). Step 2 is accomplished by identifying strongly connected components [1]. For a multiple-input operation (with associativity) in a cycle, the operands which are not in any cycle are moved outside by making a new operation to compute them. The goal of this step is to minimize the size of the recursive part by moving as many operations out of cycles as possible. To maximize the throughput, reducing the iteration bound of the recursive part of a structure is more important than minimizing the critical path. Step 3 splits the operations in cycles (operations in the recursive part) to locally minimize the iteration bound. The evaluation of iteration bound is discussed in Sec 4.1. The operations in the non-recursive part are split in order to globally minimize the critical path similar to Hartley's tree height reduction (Step 4). Constant operands are also pre-computed at this step (known as constant folding). Pipelining often follows to reduce the actual critical path to the obtained iteration bound. The main purpose of Step 4 is to minimize the number of extra pipeline stages (and hence registers).

The procedure is illustrated in Figure 2. The nodes in a dataflow graph represent operations or delays. The edges represent data transfer between nodes. The recursive part is highlighted by thick lines. The number in parentheses associated with an operand (edge) is the arrival time. The critical path of the original structure is 5 clock cycles (the edge b with the chain of 4 additions). With retiming/pipelining/loop transformations, the smallest achievable sample period is bounded by the iteration bound, which is 3. Step 1 fully collapses the chain of the 4 additions. The operands which are not in a cycle (a, b, and c) are computed with a new not-in-cycle addition (Step 2). Step 3 splits the in-cycle addition into 2 additions. The iteration bound of the transformed structure is 1. The critical path of the structure, given the iteration bound of 1, is minimized to 5 clock cycles in Step 4. After applying the pipelining, the final implementation has a critical path of 1 clock cycle at the expense of 3 extra pipeline stages.

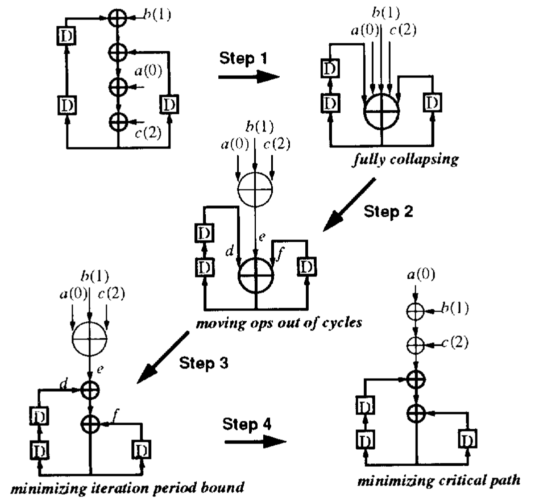

Figure 2: Illustration of associativity for minimizing iteration bound

4.1 Iteration bound evaluation

In steps 3 & 4, an operation which has more than two operands is split into binary operations. When minimizing the critical path, the arrival times of the operands are used to determine their priority in splitting. The operands which arrive earlier are computed first. When minimizing the iteration bound, the priority of the operands depends on the corresponding iteration bounds. The operands with smaller iteration bounds are computed first. An efficient way to evaluate the iteration bound is discussed in this section.

Definition 1: A ***path*** $P_{u,v}$ in a directed graph $G = (V, E)$ is a sequence of nodes ($u = v_0, v_1, v_2, \ldots, v_k = v$) such that $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $0 \le i < k$. A path is ***simple*** if all nodes in the path are distinct. A ***cycle*** C in $G = (V, E)$ is a path in which the first and last nodes are identical. A cycle is ***simple*** if $v_1, v_2, \ldots, v_k$ are distinct, otherwise it is ***composite***.

Definition 2: A ***critical path*** is a simple path which has the longest computation time (in clock cycles) and contains no delay operators. A ***critical cycle*** is a simple cycle which has the largest iteration bound.

Definition 3: Let e be an operand (edge). The ***critical cycle of order k*** with e, denoted as $C_k(e)$, is defined as a longest simple cycle passing through e and exactly k delays.

Definition 4: Let v be an operation which is in a cycle, and e be an operand of v. The ***partial length*** of $C_k(e)$, denoted as $l_k(e)$, is defined as the length of $C_k(e)$, excluding that of v.

Since the partial length $l_k(e)$ excludes the computation time of v, it is invariant to the structure of v during splitting. Let $T(v)$ be the computation time of v. The iteration bound of e is

$$IB(e) = max_k\left(\frac{l_k(e) + T(v)}{k}\right)$$

Lemma 1: Let $L^1_k(e)$ and $L^2_k(e)$ be two simple cycles passing through e and k delays. If the length of $L^1_k(e)$ is longer than $L^2_k(e)$ before splitting the operation v, $L^1_k(e)$ is still more critical than $L^2_k(e)$ after splitting.

From Lemma 1, a critical cycle going through e after splitting must be a cycle which has the longest partial length $l_k(e)$, for some order k, before splitting. Therefore, the optimal splitting for

minimizing the iteration bound can be obtained with $l_k(e)$, for all k and e, instead of considering all the simple cycles, where k is bounded by the number of delays in the graph and e refers to all the operands of the operation to split.

Let D_v be the computation time of two-input v. Assume the operation v has n operands. During the splitting, $T(v)$ is estimated as a tree structure.

$$T(v) = E_n \equiv \lceil \log(n) \rceil \times D_v$$

Definition 5: The ***predicted iteration bound*** of an operand e, denoted as ***PIB(e)***, is defined as

$$PIB(e) = max_k \left(\frac{l_k(e) + E_{n-1} + D_v}{k}\right)$$

In our approach, a multi-input operation is split incrementally by selecting two operands (out of the n) which are combined into a new operation. $PIB(e)$ predicts the bound of e in the case that e is one of the selected operands. The term $l_k(e) + E_{n-1} + D_v$ is the length of the new critical cycle passing through e and k delays. Assume v has n operands originally. After one step of splitting, v has $(n-1)$ operands. The new estimated computation time is E_{n-1}. Because every cycle going through e includes the new operation, the lengths of the critical cycles are incremented by D_v. The two operands with smaller $PIB(\cdot)$ are selected. This splitting procedure based on $PIB(\cdot)$ monotonically decreases the estimated iteration bound.

Lemma 2: Let e_1 and e_2 be the two selected operands. The partial length of the critical cycles of the newly created operand e' is

$$l_k(e') = max(l_k(e_1), l_k(e_2)) + D_v$$

To evaluate the predicted iteration bound of an operand e, one has to enumerate all the simple cycles which pass through e (to get $l_k(e)$). The best algorithm that enumerates all the simple cycles in a graph G has a time complexity of $O((|V| + |E|) * (|C| + 1))$, where $|V|$ is the number of nodes, $|E|$ the number of edges, and $|C|$ the number of cycles in the graph [8]. Since the number of cycles may grow exponentially with the number of nodes and edges, enumerating all simple cycles is not efficient.

To avoid the enumeration of simple cycles, the definition of the iteration bound of an edge e is relaxed to include *all* the cycles through e *including* composite cycles. A composite cycle, however, can pass the edge e once and the critical cycle of the graph an infinite number of times. This over-relaxation makes the iteration bound of all edges converge to the iteration bound of the graph. Some limitation on the composite cycles is, therefore, necessary. Let $|D|$ be the number of delays in the graph. A simple cycle contains at most $|D|$ delays. The iteration bound of an edge is therefore relaxed to include all the cycles which pass through e with at most $|D|$ delays. With this relaxation, if a critical cycle is a simple one, it still dominates the iteration bound of e. If all the simple cycles passing through e are not critical, the iteration bound of e is less than that of the critical cycle in the graph. In that case, the bound of the graph is not affected by e.

Gerez [4] proposed an algorithm to evaluate the global iteration bound of a recursive graph using dynamic programming (matrix multiplication). A similar idea is used in our approach for deriving the iteration bound of an edge. In the following derivation, v represents the operation to be split.

Definition 6: The ***longest path matrix of order k*** with v ($k \geq 1$), denoted by $L^k(v)$, is a $|D| \times |D|$ matrix. The entry $L^k_{ij}(v)$ is the length of the longest path from delay i to delay j passing through (k-1) delays but not v. For simplicity, we are using L^k instead of $L^k(v)$ in the rest of the paper.

Let S be the strongly connected component containing v, S_v the subgraph of S without v and all its connections, and G_v the directed acyclic graph (DAG) obtained by breaking the delays of S_v. L^1 can be obtained by solving the single source shortest path problem $|D|$ times on G_v. The length of the longest path from delay i to delay j passing through (k-1) delays, L^k_{ij}, is the maximum of $(L^1_{ix} + L^{k-1}_{xj})$, for all x. Thus L^k can be calculated by matrix multiplication over (*max*, *sum*). To evaluate the predicted iteration bound of an operand e of v, we need to calculate $l_k(e)$. Let S_e be the subgraph of S obtained by removing all operands of v, except e. Since e is contained in all the cycles in S_e and all the simple cycles containing e are in S_e, the relaxed iteration bound of e can be obtained from S_e.

Definition 7: The ***starting (ending) matrix*** with e, denoted by $\boldsymbol{U_e}$ ($\boldsymbol{W_e}$), is a $1 \times |D|$ ($|D| \times 1$) matrix. The entry u_i (w_i) is the length of the longest path from v to delay i (from delay i to v) without passing through any other delays (in S_e).

The length of the critical cycles from v to v passing through e and having (k+1) delays is the maximum of $(u_i + l^k_{ij} + w_j)$ for all i and j. Since U_e and L are defined on paths not adjacent to v, they are invariant when evaluating the iteration bounds for all operands of v. $U \cdot L^k$ does not have to be recomputed for each operand of v, where $U = U_e$. Only W_e varies with e. The partial length of the critical cycle of order k with e, $l_k(e)$, can be easily obtained by this approach ($T(v)$ is not counted in U and W_e).

$$l_k(e) = U \bullet L^{k-1} \bullet W_e$$

This algorithm solves the single source shortest path problem (on a DAG) $|D|$ times for L, once for U, and $|E_v|$ times for W_e, where $|E_v|$ is the number of operands of v. It also performs vector-to-matrix multiplication $|D|$-1 times for $U \cdot L^{k-1}$, and vector-to-vector multiplication $(|Ev| \cdot |D|)$ times for $U \cdot L^{k-1} \cdot W_e$. The complexity of this approach is $O((|D|+|E_v|) \cdot |E| + |D|^3 + |D|^2 \cdot |E_v|))$ for v.

When there are more than one multi-input operations in a strongly connected component, unlike minimizing the critical path, there is no specific way to order those operations to achieve the minimum iteration bound. The heuristic is as follows:

Rule1: The operations in the critical cycles of a strongly connected component are split first.

Rule 2: To break any tie in Rule 1, the operations with more operands are split first.

Rule 1 is easy to justify. Splitting the operations in the critical cycles can directly reduce the iteration bound of the component. For the operations which have not been split yet, the computation times are estimated as a tree structure. This estimation would be more accurate if the operation has less operands. Therefore, Rule 2 splits the operations with more operands first.

4.2 Experimental results (associativity)

Table 1 shows experimental results for the associativity algorithm. The number in each entry is the length of the critical path measured in terms of the number of clock cycles. The number in a parenthesis is the required number of pipelining stages. Since the algorithm only employs transformations which do not create new operations, the overhead in area is well bounded (see Table 2). When keeping the same sample period, the shorter critical path offers more freedom in scheduling which may reduce the number of execution units (Table 3).

TABLE 1: Critical path in clock cycles.

	---	Pipelining	Associativity+Pipelining
fir11	11	1(11)	1(4)
iir7	10	3(3)	2(6)
volterra2	12	12	4(2)

TABLE 2: Relative area (sample period = critical path)

	---	Pipelining	Associativity+Pipelining
fir11	1	2.88	4.24
iir7	1	2.37	2.40
volterra2	1	1	3.10

TABLE 3: Relative area (sample period is fixed)

	---	Pipelining	Associativity+Pipelining
fir11	1	0.66	0.59
iir7	1	0.57	0.58
volterra2	1	1	0.92

5 Enabling algebraic transformations

In this section, based on the *EP* principle, the associativity transformation (Sec 4) is augmented with the enabling algebraic transformations, distributivity and CSR, for further speed-up. These enablers cannot improve the performance directly, but eliminate the inhibiting factors which disable associativity. These enabling transformations, however, often create new operations. They should not be applied unless required (the *postponing* principle). In our approach, these enablers are applied to help remove operations from cycles (Step 2).

The condition in which the enabling transformations are applied is determined by dataflow analysis and heuristics. We briefly describe this condition with the aid of a simple example (Figure 3). Note that it is a non-linear structure. The original structure has a cycle with 2 additions, 1 multiplication, and 1 delay. The iteration bound is 3. None of the transformations on its own can reduce the bound. In this structure, it is possible to apply CSR to replicate A1 such that each operation has a single fanout, and also distributivity on (A1, M1) and (A2, M2). Instead of applying all these enablers directly, we start with associativity. If we can apply associativity on A1 and A3, one addition with ($a*b$) and c can be removed from the cycle. However, it is disabled by M1. To enable the associativity, we need to apply distributivity on A1 and M1. But the distributivity cannot be applied because A1 has 3 fanouts. CSR is required to enable the distributivity. CSR creates a redundant addition A11 and distributivity follows to move A1 and A3 together. Finally, associativity is

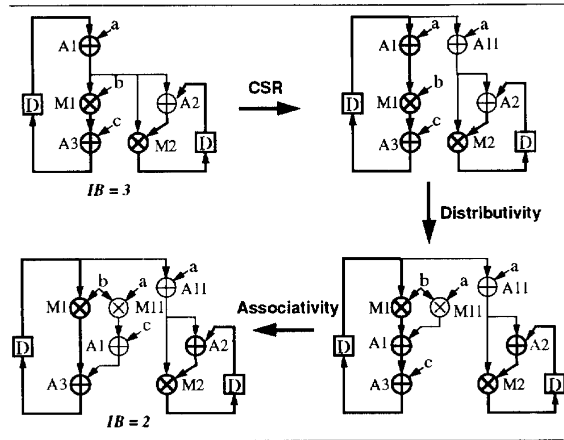

Figure 3: Illustration of algebraic transformations

applied to move one addition out of the cycle and the iteration bound is thus reduced to 2. Observe that only the necessary enablers are applied. The redundant enablers are avoided with the postponing principle. This example also shows that our approach can speed up non-linear applications. Furthermore, this approach can generally handle any distributivity pairs. For example, (max, addition) and (min, addition) are all (+, *) pairs over a commutative semi-ring. Also, multiplexing operation is distributive (as +) with other computational operations (as *)[1].

6 Example - 2nd order volterra filter

The effectiveness of the proposed approach is demonstrated with a 2nd order volterra filter. Table 4 shows the speed-up of a second order volterra filter, a non-linear recursive structure. The critical path of the structure is 12 clock cycles. Pipelining alone cannot reduce the critical path because the iteration bound is also 12. With associativity and extra algebraic transformations, the critical path can be reduced to 4 and 2 clock cycles respectively.

TABLE 4: Critical path in clock cycles.

	---	Pipelining	Associativity +Pipelining	Algebraic +Pipelining
critical path	12	12	4	2

Since a volterra filter has only linear recurrences, an arbitrary speed-up can be achieved with the help of (time) loop unrolling. This is shown in Table 5. When a loop is unrolled with a factor of N, it processes N samples per period. The equivalent critical path per sample is the critical path divided by N. Since the critical path is constant while the unrolling factor increases, the speed-up is linear with respect to the unrolling factor. The last two rows in the table show that the overhead in area is slightly more than linear with respect to the speed-up (bounded between the AT and AT^2, where A stands for area and T for normalized equivalent critical path).

1. Determining exactly when to fire an enabling transformation is out of the scope of this paper. For a more detailed description, we refer to [7].

TABLE 5: Linear speed-up of a 2nd order volterra filter

unrolling factor	1	2	4	8	16
critical path	2	3	3	3	3
equiv. CP	2	1.5	0.75	0.375	0.1875
speed up	1	1.33	2.67	5.33	10.67
relative area	1	1.75	4.22	9.41	19.86
AT	1	1.31	1.58	1.76	1.86
AT^2	1	0.98	0.59	0.33	0.17

7 Automated framework

In this section, the methodology to automatically optimize a dataflow graph to satisfy the given throughput requirement is presented. Suppose that the given sample period is less than the iteration bound. Associativity is applied to minimize the bound first. If the sample period cannot be met, the enabling algebraic transformations are activated. In case that algebraic transformations are still not sufficient, loop unrolling is used to exploit extra parallelism to further enable the algebraic transformations. The unrolling factor is determined by the min bound (assuming linear speed-up can be achieved). This is an iterative approach. Whenever the constraint is satisfied, the retiming/pipelining is applied to reduce the critical path to meet the given sample period. This approach is deterministic with low complexity.

This methodology is demonstrated with a 3th order lattice filter with the given sample period of 1 clock cycle. The sequence of transformations to be applied is shown in TABLE 6. The iteration bound of the original structure is 6. Associativity alone cannot reduce the bound (step 1). Using algebraic transformations, the bound is reduced from 6 to 3 (step 2). Since the sample period is only 1 clock cycle, the minimum unrolling factor is 3 (step 3). Enabled by loop unrolling, algebraic transformations successfully reduce the iteration bound to 1 (step 4). The last step is to apply pipelining to reduce the critical path to the sample period which is 1 clock cycle.

TABLE 6: Automatic transformations for third order lattice filter.

Step	Transformation	CP	Eqv. CP	IB	Eqv. IB
---	---	14	14	6	6
1	Associativity	13	13	6	6
2	Algebraic	7	7	3	3
3	Unroll (3)	13	4.3	9	3
4	Algebraic	12	4	3	1
5	Pipelining	3	1	3	1

8 Conclusion

A new ordered transformational framework, based on the *enabling and postponing principle*, has been proposed. It is observed that reducing the iteration bound is the real target behind performance optimization. With the enabling principle, we derived a *deterministic* ordering of the transformations for *iteration bound reduction*. The postponing principle is to simultaneously and indirectly minimize area while optimizing speed. This framework can automatically transform both linear and nonlinear computational structures to improve the throughput. We also presented novel algorithms to apply the associativity and algebraic transformations to minimize the iteration bound (with the *EP* principle). This framework has been implemented in the HYPER system[18]. Since this approach has a limited hardware overhead, it enables the speed-up of many real-life DSP numerical computations. In addition, this framework establishes an environment for algorithmic design space exploration.

9 Acknowledgements

This project was sponsored by Semiconductor Research Cooperation (93-DC-008) as well as MICRO.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest: "Introduction to algorithms," *The MIT Press*, 1992.

[2] G. Fettweis, L. Thiele, H. Meyr: "Algorithm transformations for unlimited parallelism," *ISCAS*, vol. 2, pp. 1756-1759, May 1990.

[3] G. Fettweis, L. Thiele: "Algebraic recurrence transformations for massive Parallelism," *VLSI Signal Processing V*, pp. 332-341, 1992.

[4] S.H. Gerez, S.M.H. de Groot, O.E. Herrmann: "A polynomial-time algorithm for the computation of the iteration-period bound in recursive data-flow graphs," *IEEE Trans. on Circuits and Systems-I: Fundamental Theory and Applications*, vol. 39, no. 1, pp. 49-52, Jan 1992.

[5] R. Hartley, A. Casavant: "Tree-height minimization in pipelined architectures," *ICCAD*, pp. 112-115, Nov. 1989.

[6] L. Hyafil, H.T. Kung: "The complexity of parallel evaluation of linear recurrences," *JACM*, vol. 24, no. 3, pp. 513-521, July 1977.

[7] S.-H. Huang, "Behavioral Transformations for High Performance Real Time Applications to Maximize the Throughput." *Master report*, University of California at Berkeley, May 1993.

[8] D. Johnson: "Finding all the elementary circuits of a directed graph," *SIAM J. Computers*, vol. 4, no. 1, pp. 77- 84, March 1975.

[9] P.M. Kogge: "Parallel solution of recurrence problems," *IBM J. Res. Develop.*, vol. 18, pp. 138-148, March 1974.

[10] H.T. Kung: "New algorithm and lower bounds for the parallel evaluation of certain rational expressions and recurrences," *JACM*, vol. 23, no. 2, pp. 252-261, April 1976.

[11] C.E. Leiserson, F.M. Rose, J.B. Saxe: "Optimizing synchronous circuitry for retiming," *3rd caltech Conference on VLSI*, pp. 87-116, March 1983.

[12] D.G. Messerschmitt: "Breaking the recursive bottleneck," *Performance Limits in Communication Theory and Practice, Kluwer Academic Publisher*, 1988.

[13] K.K. Parhi, D.G. Messerschmitt: "Pipeline interleaving and parallelism in recursive digital filters - part I: pipelining using scattered look-ahead and decomposition," *IEEE Trans. on ASSP*, vol. 37, no. 7, pp. 1099-1117, July 1989.

[14] K.K. Parhi, D.G. Messerschmitt: "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding." *IEEE Trans. on Computers*, vol. 40, no. 2, pp. 178-191, Feb 1991.

[15] M. Potkonjak, J.M. Rabaey: "Fast Implementation of Recursive Programs Using Transformations," *ICASSP*, March 1992.

[16] M. Potkonjak, J.M. Rabaey: "Maximally fast and arbitrarily fast implementation of linear computations," *ICCAD*, pp. 304-308, Nov. 1992

[17] M. Potkonjak, J.M. Rabaey: "Pipelining: Just another transformation," *Int'l Conf. on Application Specific Array Processors*," pp. 163-175, 1992.

[18] J.M. Rabaey *et al.*, "Fast Prototyping of Datapath-Intensive Architectures," *IEEE Design & Test of Computers*, pp. 40-51, June 1991.