3-1-1999

A Computational Redundancy Reduction Approach for High performaice and Low Power DSP Algorithm Implementation

Khurram Muhammad
Purdue University School of Electrical and Computer Engineering

Kaushik Roy
Purdue University School of Electrical and Computer Engineering

A Computational Redundancy Reduction Approach for High Performance and Low Power DSP Algorithm Implementation

Khurram Muhammed
Kaushik Roy

TR-ECE 99-3
March 99

School of Electrical
and Computer Engineering
Purdue University
West Lafayette, Indiana 47907-1285

A Computational Redundancy Reduction Approach for High Performance and Low Power DSP Algorithm Implementation[1]

Khurram Muhammad and Kaushik Roy
Email: khurram@ecn.purdue.edu, kaushik@ecn.purdue.edu
School of Electrical and Computer Engineering,
Purdue University, West Lafayette, IN 47907

February 22, 1999.

Abstract

In this paper, we present a general approach which specifically targets reduction of redundant computation in common *digital signal processing* (DSP) tasks such as filtering and matrix multiplication. The main idea presented in this work is to show that such tasks can be expressed as multiplication of vectors by scalars and fast multiplication can be achieved by sharing computation in such operations. The multiplication schemes considerably reduce redundant computation by decomposing the vectors in a manner which results in maximal computation sharing, thereby, resulting in a faster and potentially low-power implementation. Two decomposition approaches are presented, one based on a *greedy decomposition* and the other based on *fixed-size lookup* rule which lead to two multiplication architectures for scaling of vectors. Analysis of the proposed implementations shows a speed-up by a factor of up to **3** over a carry save array multiplier. Analog simulation of an example 8-bit multiplier shows a speed advantage by a factor of 1.85 and a power disadvantage of 1.9 over a conventional carry save array multiplier. Using voltage scaling, the power consumption of the example multiplier can be reduced to 56% of the carry save array multiplier.

'This work was supported in part by DARPA (F33615-95-C-1625), NSF CAREER award (9501869-MIF'), Rockwell, AT&T and Lucent foundation.

I. Introduction

The ever increasing demand for services and mobility in communication and computing: places increased challenges in the design of such systems. New techniques and approaches are required at all levels of design abstractions as future technologies are expected to provide unprecedented levels of computational performance in small hand-held units. Since the evolution in battery technology has not yet caught up with the demands in computational requirements, it provides us with a motivation to consider new approaches to reduce computation without compromising the constraints on system performance. The classical approaches for reducing complexity of high-performance *digital signal processing* (DSP) comprise the use of techniques such as *recursion* (e.g. RLS, FFT algorithms), *multi-rate signal processing* and *low rank approximation*. The last technique is a widely used approach which compromises accuracy by removing less significant computations from a given computational algorithm. Low-complexity design not only improves the speed at which the algorithm can process data, but it also leads to low power design at the highest level of abstraction by reducing energy consuming operations.

In this paper, we explore complexity reduction from the point of view of reducing the overall number of operations in common DSP tasks such as filtering and matrix multiplication by using the concept of *computation sharing* multiplication. The main insight provided to the subject of low-complexity design is the reduction of *computational redundancy* [1] which is defined as the *excess computation over the minimum number of bit operations needed for a given sequence of operations*. The basic idea is to speed-up computation by identifying common computations in a sequence of operations, and to investigate and propose structures which achieve that objective with minimal computational overhead. The structures proposed in this work which achieve this goal are multipliers which can be viewed as generalized higher-order coded multipliers. Since, computation reduction is achieved through *computation sharing*, no quantization loss in incurred using the proposed multiplication technique.

Many multiplier structures [2], [3], [4], [5] have been investigated and proposed during the past four decades. However, these architectures do not assume existence of any relationship between successive computations and provide gains in execution speed for any given operand without regards to the past or future input. The operations in DSP tasks are generally based on data sequences (or data vectors) and gains in speed of operation can be achieved by identifying and exploiting relationships between numbers in the given sequence. The primary difficulty in such an approach is that it requires additional computational overhead to identify such relationships [1] and such an effort can only be justified in systems such as non-adaptive filters where this computation can be performed off-line [1]. Secondly, extra memory overhead may be required to store intermediate results since the order of computation may be altered [1], [6]. In the situation where data values are always changing, simple schemes must be explored such that they can identify computation which can be shared amongst a sequence of operations without incurring computational overhead for locating such relationships.

The main idea presented in this paper is to decompose multiplication operations in terms of addition of shifted values of *alphabets*. An alphabet can be viewed as a number in a higher radix representation scheme The best alphabets for various scenarios of vector scaling operations are investigated and proposed. Using these alphabets, we show the relative computational improvements which are possible if efficient implementations of multiplication based on these alphabets are designed. We propose such implementations and analyze their performance showing that a substantial speed advantage is possible by using such a multiplication scheme. The major contribution of this paper are summarized below:

- We identify the relavence of increasing the speed of vector scaling operation with the common DSP tasks such as filtering and matrix multiplication.
- We identify simple number decomposition strategies and identify the *optimal* alphabets for general input vectors.
- We show the simple unit comprising a multiplexor (MUX) and two shifters (SHIFT) is required to implement the proposed multiplication which is based on decomposition of inputs to alphabets.
- The proposed multiplier structures are analyzed in detail for their speed performance and compared with a *carry save array* (CSA) multiplier. Further, the trade-offs in their implementation aspects are investigsted.

The rest of the paper is organized in seven sections. Section II provides a review of basic DSP tasks and shows their relationship with vector scaling operation. Section III describes the representation of computations in vector scaling operation in terms of a graph. Section IV describes the basic approach employed in reducing computational redundancy using the graph and presents the decomposition rules. Section V describes the resulting constrained optimization problem which is solved to obtain optimal alphabets using an exhaustive search. The optimal alphabets obtained and the computational advantage due to propcsed technique are shown in section V-B. In section VI, we describe various ways to implement the proposecl multiplication scheme and analyze the performance of the proposed architectures. Finally, section VII concludes this paper.

II. General Background

This section provides a review of the most common operations in general DSP tasks. Consider the fundamental operation of multiplication of a given vector by a scalar quantity. This can be expressed as $y = c \cdot x$, where both y and $\boldsymbol{c}$ are column vectors of size M and x is a scalar quantity. The main reason for considering this operation as a fundamental operation in DSP applications is the realization that the operand x is shared by all the elements of c and therefore, computation can be shared as explained in later section:;. This operation requires M multiplication operations. We next show that this operation is fundamental to most DSP tasks such as filtering and matrix multiplication.

A. Digital *Filtering*

Consider a linear time-invariant (LTI) FIR filter of length M described by an input-output relationship of the form $y(n) = \sum_{i=0}^{M-1} c_i\ x(n - \mathrm{i})$. In this context, c_i represents the ith coefficient and $x(n - \mathrm{i})$ denotes the data sample at time instant $n - \mathrm{i}$. Figure 1(a) shows a parallel implementation of an 8-tap (M = 8) FIR filter with symmetric coefficients. A single, shared multiplier based implementation is shown in figure 1(b). Since the goal is to compute $y(n)$ as fast as possible, a shared multiplier based implementation holds coefficients and data in separate memories and applies these to the input of a *multiply and* accumulate (MAC) unit. The output $y(n)$ is computed after M multiplication operations. We notice that such an implementation computes an inner product in which all elements are distinct and no opportunity of computation sharing arises. Hence, the only approach to speed up this computation is to employ pipelining or faster multipliers. If we allow block processing in which the output is computed

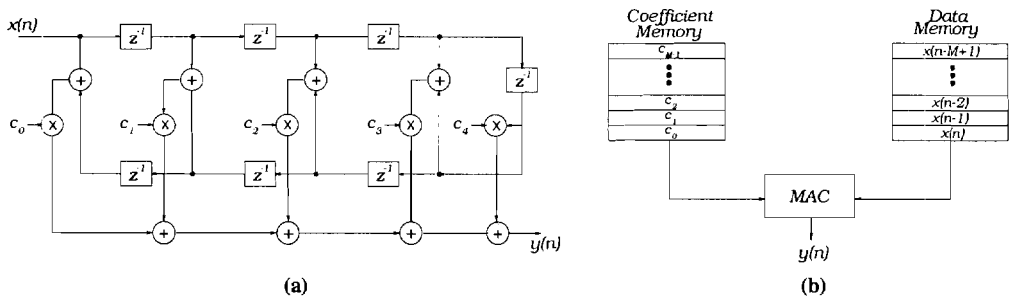

Fig. 1. Implementation of an 8-tap filter. (a) Parallel implementation, and (b) Shared multiplier based implementation.

as blocks of size N, we can represent the output by a column vector y of size N. The modified equations for FIR filtering can be expressed as

$$\mathbf{y}(n) = \mathbf{X}(n)\mathbf{c} \tag{1}$$

$$= \mathbf{C}\tilde{\mathbf{x}}(n) \tag{2}$$

where $\mathbf{y}(n) = [y(n), y(n-1), \ldots, y(n - \mathrm{N} + 1)]^T$ and $\mathbf{X}(n)$ is a Toeplitz data matrix of dimension $\mathrm{N} \times \mathrm{M}$ such that the ith column of X contains the elements $\mathbf{x}(n-\mathrm{i}) = [x(n-\mathrm{i}), x(n-\mathrm{i}-1), \ldots, x(n-\mathrm{i}-\mathrm{M}+1)]^T$ for $\mathrm{i} = 0, 1, \ldots, \mathrm{N} - 1$. The coefficients appear as a column vector $\boldsymbol{c} = [c_0, c_1, \ldots, c_{M-1}]^T$. The coefficient matrix $\boldsymbol{C}$ in equation 2 is of dimension N x (N + M + 1) such that the ith row consists of i zeros followed by the elements of vector $\boldsymbol{c}$ followed by $(\mathrm{N} + 1 - i)$ zeros for $\mathrm{i} = 0, 1, \ldots, N - 1$. The vector $\tilde{\mathbf{x}}(n) = [x(n), x(n-1), \ldots, x(n - \mathrm{N} - \mathrm{M} + 1)]$. Then, equation 2 can be written as $\mathrm{y} = c_0\mathbf{x}(n) + c_1\mathbf{x}(n - 1) + \ldots + c_{M-1}\mathbf{x}(n - \mathrm{M} + 1)$ Since c_i's are scalar quantities, the above equation recasts the block filtering problem using the fundamental operation of vector scaling. One can similarly show that equation 2 can also be expressed in terms of vector scaling operations. Similarly, it can also be verified that the output equation of a block LTI *infinite* impulse response IIR can also be expressed in terms of vector scaling operations. Hence, vector scaling operation can be considered as a fundamental operation in digital

filtering.

B. Matrix Multiplication

The most common DSP tasks constitute multiplication of a matrix with a scalar, a vector, and another matrix. The first can be trivially expressed in terms of vector scaling operation. The second type of operation was considered in the previous section. Consider the multiplication of two matrices **A** and B of dimensions N x M and M x P, respectively. Let $A = [\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{M-1}] = [\bar{a}_0^T, \bar{a}_1^T, \ldots, \bar{a}_{N-1}^T]^T$ where $\mathbf{a}_i$ and $\bar{a}_j$ represent the ith column and jth row of **A**, respectively. The column vectors of B and D are similarly defined. Then the columns and rows of the product D = AB can be expressed, respectively, as $\mathbf{d}_i = \sum_{m=0}^{M-1} \mathbf{a}_m b_{m,i}$ and $\bar{d}_j = \sum_{m=0}^{M-1} a_{j,m} \bar{b}_m$ for $i = 0, 1, \ldots,$ **P** and $j = 0, 1, \ldots, N$. Clearly, multiplication of two matrices can be expressed in terms of vector scaling operations. The above two interpretations yield identical end result, however, the number of elements in the vectors which need to be scaled depends on the ratio N/P. Since, larger vectors irnply a larger potential for sharing of computation and our goal is to maximize sharing to reduce redundant computation, we consider the interpretation which results in larger vector sizes. Hence, the first interpretation is favored for $N > P$, whereas the second one is favored when $N <$ **P**. Any of the two may be used for the case $N = P$.

III. Representation of Computation

We now turn our attention to the vector scaling operation. Since this operation requires two operands, there are only two possible scenarios of interest. First, both are variable, and second, one is fixed and the other variable. The trivial situation of both operands being fixed is not interesting because one can pre-compute the result and eliminate such operations. If only one operand is fixed, either the vector has variable entries and the scalar multiplier is fixed, or the vector has fixed elements and the scalar is variable. We will refer to the prior situation as VVFS (Variable Vector, Fixed Scalar) situation and the latter one as FVVS (Fixed Vector, Variable Scalar) situation. We note that in many situations it may be possible to specify the problem as either VVFS or FVVS. This can be seen for the non-adaptive filtering example in equations 1 (VVFS) and 2 (FVVS) by choosing the vector to be composed of elements of input data or the filter coefficients, respectively. In case of adaptive filtering, both the vector arid the scalar are variable and this condition is referred to as *VVVS* (Variable Vector. Variable Scalar) condition.

In order to reduce computational redundancy, the first step is to express the desired operation in terms of vector scaling operations. We will refer to this step as *computation reordering.* The main goal of this step is to transform computation in the given DSP task in a frame-work where efficient computation sharing techniques are known. In this paper, we do not specifically address techniques for such computation reordering in general DSP tasks, however, examples of this step have already been presented in sections II-A and II-B. The main emphasis of this paper is to address the second step, in which efficient computation sharing approaches are investigated for the fundamental operation of vector scaling. As shown in earlier

sections, the most common DSP tasks can be expressed in terms of this fundamental operation, and hence, the computation in a given DSP algorithm can be represented in terms of the representation of computation in the the vector scaling operation.

One approach in representing the computation in vector scaling operation is to construct a complete graph $G = (V, E)$, where V and E represent the sets of vertices and edges, respectively. The elements in V are the M elements of the given vector, and the edge $e_{i,j}$ between vertex v_i and v_j expresses the effort of computing the product $v_j s$ in relation to $v_i s$, where s represents the scalar operand. $e_{i,j}$ will always have a positive value. There may be more than one edges between v_i and v_j depending on how many ways v_j may be computed given v_i has already been computed [1]. Figure 2 shows the graph for $M = 4$. The interpretation of $e_{i,j}$ depends on how a computation is viewed for a particular "style" of implementation. The word *style* has a broad meaning and includes architectural style and technological coiisiderations. For example, in a fully parallel implementation, *shifts* of values can be implemented using wires and hence, if v_j can be obtained from v_i by a simple shift operation, the computation $v_j s$ can be simply olbtained "free" of computational cost from the prior computed $v_i s$ for some scalar s. Hence, the particular edge $e_{i,j}$ for such implementation is 0. In contrast, if this computation was performed without consideration to its previous occurance it would be a redundant computation. In summary, by proper definition of vertices and edge

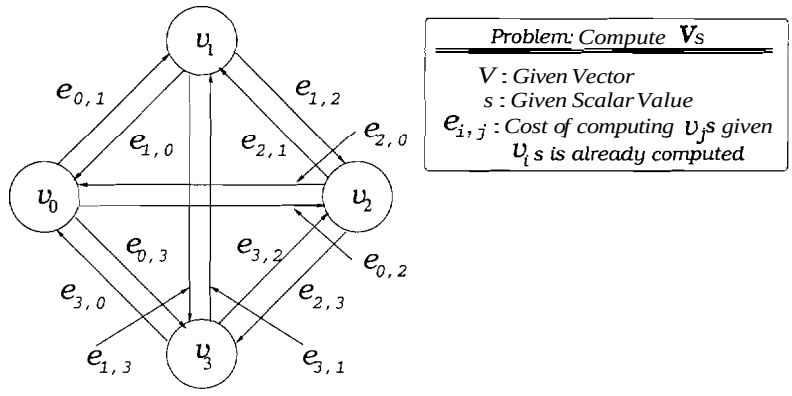

Fig. 2. Graph representation.

weights, the graph representation can adequately express relationships between different computations. By proper identification of these inter-relationships, a strategy for reordering and sharing computation may be formulated thereby obtaining implementations with low computationally redundancy, lower execution time and lower power dissipation.

A. Decomposition of Computation in Vector Scaling

Consider the vector scaling operation $V s$ in which $V = \{v_0, v_1, \ldots, v_{M-1}\}$ and s is a given scalar. Now, our goal is to identify operations that can be shared between the products $v_i \,.\, s$, $i = 0, 1, \ldots, M - 1$ *maximally*. The first step is to decompose each vertex into *alphabet sets*. An *alphabet set* is a set of

alphabets which can represent the given vertex using *add* and *shift* operations. An *alphabet* is a value which can be used to write the decomposition of the given vertex. Figure **3** elaborates this idea. Given a vertex, v_i, it is possible to construct all possible alphabets sets $\bar{\alpha}_0, \bar{\alpha}_1, \ldots, \bar{\alpha}_R$ such that each set contains alphabets which can completely represent v_i. That is, for $j = 0, 1, \ldots, \mathrm{R}$,

$$v_i = \sum_{k=0}^{Lj} 2^{m_k} \alpha_{k,j} \tag{3}$$

where m_k ace the appropriate *shift* values required to correctly obtain v_i. In the above equation, $\alpha_{k,j}$ for $\mathrm{k} = 0, 1, \ldots, L_j$ are alphabets belonging to the alphabet set j. This decomposition requires $L_j - 1$ *add* and *shift* operations. To understand this decomposition, consider the example in figure **3** in which all four (R = **3)** possible alphabet sets for v_i = 10011 are listed. The alphabets in these sets are $\alpha_{0,0} = \alpha_{0,1} = \alpha_{1,2} = 1$, $\alpha_{0,2} = 1001$, $\alpha_{1,1} = 11$ and $\alpha_{0,3} = 10011$. For the set, $\bar{\alpha}_0 = \{\alpha_{0,0}\} = \{1\}$, we can write $v_i = 2^4\alpha_{0,0} + 2^1\alpha_{0,0} + 2^0\alpha_{0,0}$. This decomposition requires two *add* and *shift* operations. Note that $Lo = 1$, $m_0 = 4, m_1 = 1$ and $m_2 = 0$. For the remaining three sets, we have $v_i = 2^4\alpha_{0,1} + 2^0\alpha_{1,1}$, $v_i = 2^1\alpha_{0,2} + 2^0\alpha_{1,2}$ and $v_i = 2^0\alpha_{0,3}$ for the second, third and fourth sets, respectively. The size of the alphabet sets are $L_1 = L_2 = 2$ and $L_3 = 1$. In the sequel, we will refer to the computation of the form $2^{m_k}\alpha_{k,j}$. *s* as a *sub-computation.* Hence, each alphabet set gives rise to a different scheme of adding sub-computations to obtain v_i . *s* and maximal computation sharing results when we use an alphabet set which minimizes the overall number of distinct subcomputations required to compute Vs.

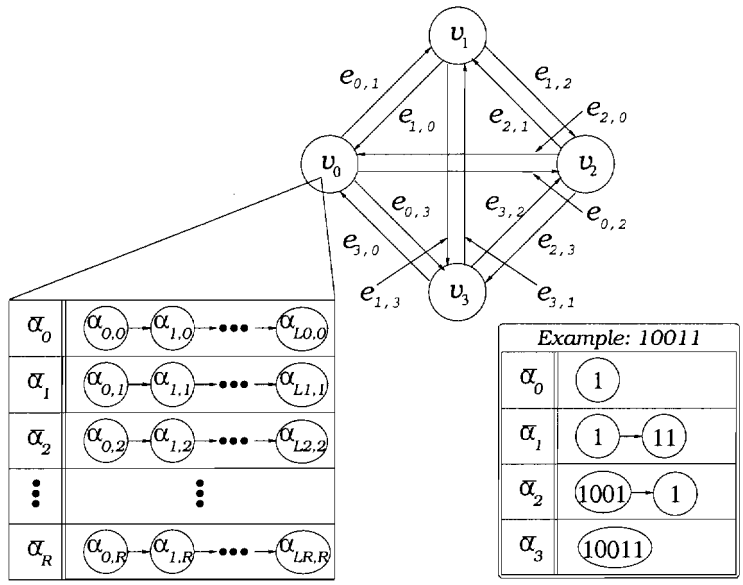

Fig. **3.** Alphabet Sets and Their Relationship to the Graph.

Now each vertex $v_i \in \mathrm{G}$ for $i = 0, 1, \ldots, \mathrm{M} - 1$ can be similarly decomposed into R_i alphabet sets. For a given vertex v_i, each alphabet set is one way to *cover* the vertex and in order to obtain the product $v_i \cdot s$, we need all the alphabets in *at least* one alphabet set for the vertex. This is evident if both sides of equation

3 are multiplied by s. Hence, alphabets in an alphabet sets can be viewed as nodes connected in series and an alphabet sets can be viewed as *parallel* paths through which $v_i \cdot s$ can be obtained. Each vertex in G can be viewed as a *super-node* containing series-parallel paths of connected nodes .and the vertex is said to be *covered* if all the series nodes in *at least* one alphabet set are visited. There are R_i possible ways to cover the vertex v_i. In terms of graph representation, on selecting an alphabet set which covers a vertex v_i, an edge $e_{i,j}$ expresses two metrics. 1) The number of *add* operations required to compute v_i through the alphabets in the selected alphabet set using equation 3, and, 2) The number of new alphabets introduced in the solution alphabet set due to the choice of the selected alphabet set. Hence, edge weights are *dynamic* and reflect both these quantities independently. Since, there are R_i choices of alphabet set selection at node v_i, there are exactly R_i possible edge weights that may be assigned to an edge $e_{i,j}$ that emanates from v_i. Our next objective is to minimize the the cost of a *tour* in G such that the number of alphabets in the *solution alphabet set* is minimum. We next describe the multiplication strategy which arises as a result of the considerations outlined in this section.

IV. Removing Computational Redundancy - The Basic Approach

In order to obtain *maximal sharzng* in the computation of $V s$, we need to identify a *solution alphabet set* of alphabets which satisfies the following properties.

1. It covers all the vertices in G.
2. It minimizes the number of alphabets in the set, and,
3. The total number of *add* operations in the decompositions of all vertices, v_i, $i = 0, 1, \ldots, M-1$, is minimized.

This can be mapped to the classical NP-complete problem known as the *weighted minimum set cover* (WMSC) [7]. We assume that a "good" solution set can be computed using any of the known heuristic based techniques which gives the solution set $\bar{\alpha}_s = \{\alpha_0, \alpha_1, \ldots, \alpha_{Q-1}\}$ comprising of Q alphabets. However, we must first consider the implementation aspects of the resulting multiplier structure in order to define a multiplication strategy which is practical and simple to implement.

A. Cornputation Sharing Multiplication – General Structure

The multiplier must provide an ability to decompose an applied input v_i into alphabets in the solution alphabet set, be able to perform the sub-computation $\alpha_k \cdot s$ where $\alpha_k \in \bar{\alpha}_s$, and must also possess the ability to provide the *shift* necessary to correctly re-compose the final product $v_i s$ from the sub-computations. That is, it must compute

$$v_i s = \sum_{k \in Q} 2^{m_k} \alpha_k \cdot s \tag{4}$$

where m_k's are appropriate shifts. Note that $\bar{\alpha}_s$ may contain more alphabets than are necessary to cover a given v_i as its elements must be able to cover all the elements in V

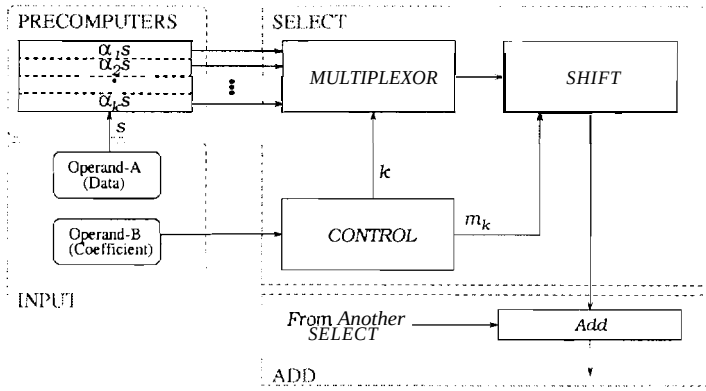

Fig. 4. Basic Structure of the Multiplier.

Figure 4 shows the basic structure of the resulting multiplication strategy. The multiplier consists of a bank of Q + 1 parallel multipliers which compute the products $\alpha_k \cdot s$ for k = 0, 1, . . . , Q. We will show in the later section that these multipliers can be replaced by simple adders. We will refer to these units as *PRECOMPUTERS* as their function is to pre-compute the sub-computations which are to be shared. Note that the computation performed by the bank of PRECOMPUTERS is required only once during the M product generations in the vector scaling operation. A CONTROL unit decomposes the applied value of v_i into the form of equation **3** and sends the appropriate *index* signal to a MUX which selects the correct $\alpha_k \cdot s$ sub-computation. Since the CONTROL unit is used to decompose v_i, it already knows the shift value (m_k) in the decomposition, hence it can send the *shift* value to the SHIFT unit. The shifted value of the selected product is then added with the next sub-computation in equation 4.

The number of alphabets in the solution set determine the number of PRECOMPUTERS in the bank. It also dictates the complexity of the MUX unit which must select the appropriate pre-computed product. The complexity of the SHIFT unit is determined by the word-length, W, of elements in **V** and the length of the longest alphabet in the solution set. For example, if W = 8 and the longest alphabet in solution set is 1111, then the SHIFT unit must be able to shift 8+4 = 12 outputs of the selected PRECOMPUTER by a maximum of W – 1 bits. The function of the CONTROL unit is to decompose an applied input into the form expressed in equation **3.** The complexity of this unit is primarily determined by the decomposition strategy used to obtain the form expressed in equation **3.** We first observe that there is no unique way to express a given number in the form in equation **3.** The optzmal decomposztion strategy for a given number v_i may be defined as the scheme which minimizes L_j in equation **3.** Hence, optimal decomposition of v_i decomposes it into the smallest possible number of alphabets from a given solution set, thereby resulting in the minimum number of sub-computations in equation 4. Furthermore, optimal decomposition rule requires partitioning of the set of elements in V into the minimum number of total partitions which begin and end with 1's (also called alphabets). This is not only difficult to implement, but it may also prove to be computationally expensive thereby nullifying the savings in computation due to sharing. This is

because the optimal decomposition rule is the solution to WMSC problem and the solution set obtained by solving this problem may require searching for "difficult" patterns in the inputs. Hence, the complexity of the multiplication problem simply shifts to pattern recognition problem. Therefore, we next consider the rules of number decomposition for a simple and effective decomposition scheme as this governs the complexity of the CONTROL unit.

A.1 Decomposition Rules

To decompose the applied input in a way which selects minimal number of *add* operations, we need a strategy which is simple to implement. A practical alternative is to resort to a simpler greedy strategy in which the input is scanned in a given direction starting at a given point and the longest matching alphabet in $\bar{\alpha}_s$ is identified. Suppose, the scan is performed from the *least significant bit* (LSB) towards the *most significant bit* (MSB). Then the greedy selection rule consists of two steps. 1) To minimize the total number of partitions (alphabets), we *select the longest matching alphabet from the solution set* (local minima), and, 2) Any sequence of consecutive zeros separating two alphabets is discarded. This is later corrected by the SHIFT unit. This decomposition rule will be referred to as the *greedy decomposition* (GD) rule. Since two scan directions are possible, we will refer the MSB to LSB scan rule as $GD_{M \rightarrow L}$, and the LSB to MSB scan rule as $GD_{L \rightarrow M}$.

Consider the $GD_{L \rightarrow M}$ rule. Each element in V is applied at the input of the multiplier. The input is scanned from LSB towards the MSB discarding all zeros it encounters before the first "1". At that point, it determines the longest alphabet in the solution set which matches the bits in the input. Again sequence of consecutive zeros is discarded before encountering the next "1" after the identified alphabet. Once again, the unit determines the longest alphabet in the solution set which matches the input at this point. The process is repeated until the entire input number is decomposed into alphabets from the solution set. The $GD_{M \rightarrow L}$ rule is implemented similarly with the scan direction reversed. The reader is cautioned that although this scheme seems to dictate a complex search procedure, in reality, the structure of the CONTROL for the optimal alphabets for the GD strategy turns out to be a simple *shifter*. Figure 5 shows the basic approach used in the decomposition. The shaded parts of the inputs represent sequence of zeros separating alphabets which are discarded during the scan. The number of zeros in these are used to form the *shift* signal to the SHIFT unit to ensure that the sub-computations are formed using the correct amount of shifts (m_k's in equation 4). Note that the alphabets are labeled identically in both $GD_{L \rightarrow M}$ as well as $GD_{M \rightarrow L}$, even though $GD_{M \rightarrow L}$ encounters the bits comprising the alphabet in the reverse sequence.

Let us elaborate the structure of the multiplier and rules of decomposition using an example. Let us suppose that the best alphabet consists of the set $\bar{\alpha}_s = \{1, 11, 101, 111\}$. Then the products $1 \cdot s = s$, $11 \cdot s$, $101 \cdot s$ and $111 \cdot s = (1000 - 1) \cdot s$ can all be formed using three parallel *add* operations. Hence, the bank of PRECOMPUTERS is simply a bank of three parallel adders. These sub-computations with

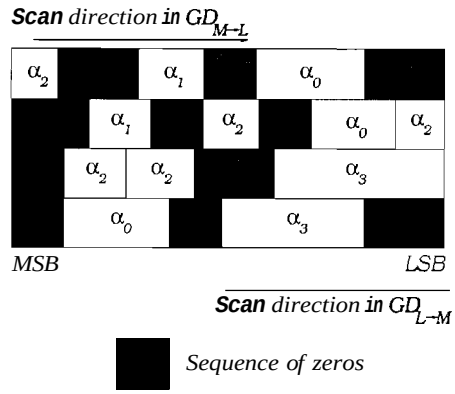

Fig. 5. Decomposition of Input Values into Alphabets.

the alphabets are computed only once for the next M products. Nest, suppose that we apply an input $v_i = 11101001$. In order to uniquely decompose the input, we scan v_i in a fixed direction. Let us consider using the $GD_{L\to M}$ rule. Then v_i is uniquely decomposed in to $11 \cdot 2^6 + 101 \cdot 2^3 + 1 \cdot 2^0$, i.e. the scanner will match 1, followed by 101 which is again followed by 11. The sequence of zeros separating the alphabets is simply ignored, however, the number of zeros in the sequence is counted to properly compute the *shift* values of $6, 3$ and 0 for $11, 101$ and 1, respectively. The three sub-computations which must be selected by the MUX are $11 \cdot s$, $101 \cdot s$ and $1 \cdot s$, with the corresponding shift values of $5, 3$ and 0, respectively.

The corresponding alphabet set for $GD_{M\to L}$ rule are the alphabets $\{1, 11, 101, 111\}$ scanned in the opposite direction, i.e. the set $\{100, 110, 101, 111\}$. This rule will decompose the input as $111 \cdot 2^5 + 100 \cdot 2^1 + 100 \cdot 2^{-2}$. The main advantage of $GD_{M\to L}$ appears when we observe that $111 \cdot 2^5 + 100 \cdot 2^1 + 100 \cdot 2^{-2} = ((111 \cdot 2^{3+1} + 1) \cdot 2^{3+0} + 1) \cdot 2^{-2}$. We note that the decomposition can be expressed in terms of relative shifts of alphabets with respect to each other, while scanning from MSB towards LSB. The computation can proceed in the order indicated by the nesting of parenthesis, with each subsequent sub-computation providing a fixed alignment of 3 (i.e. length of the alphabets) plus the number of zeros separating the alphabets. In contrast, $GD_{L\to M}$ computes the sub-computations in the reverse direction, and hence, it cannot take advantage of this observation. The final shift of -2 is a conceptual aid to clearly see the symmetry with the $GD_{L\to M}$ rule which assumes zeros leading the MSB. As explained further in section VI-A, $GD_{M\to L}$ rule simplifies the structure of the SHIFT unit. Note that GD multiplication scheme can be viewed as:

- A generalized higher order coded multiplication scheme. Both normal and Booth multiplications can be viewed as a special cases of this scheme.
- A *hybrid* look-up table based multiplication in which both look-up and computation are performed to reduce computation in multiplication. The look-up table is compressed with elements which result in minimum number of add operations required to compute the product. Computation is performed in adding the sub-computations.

As shown in figure 4, the CONTROL-MUX-SHIFT unit can be lumped into a SELECT unit, which performs the decomposition, selects appropriate product and yields correct sub-computation by providing appropriate amount of shift to the selected value. The complexity of this unit depends on the selection of alphabets in the solution set and the size of the solution set. The delay through the SELECT unit appears in series with the ADDER and increases the length of the critical path. Hence, a simpler SELECT unit facilitates design of a faster multiplier as well as reduces the power consumption in the overhead. In section VI we discuss in detail the specifics of various multiplier structures for vector scaling operation and analyze their performance. We first consider the computation reduction potential using the proposed approach.

V. The Constrained Minimization Problem

We summarize all the implementation constraints discussed in the previous sections to formulate a constrained minimization problem. Our objective is to find a solution alphabet set such that:

1. The alphabets in the solution set cover all the vertices in G.
2. The solution set is of minimal size or cardinality.
3. The total number of *add* operations in the decompositions of all vertices, v_i, i $= 0, 1, \ldots, M-1$, is minimized.
4. The number of alphabets in the solution set is a given power of two.
5. The applied input v_i is scanned for decomposition using GD rule.

The simplest way to impose the first condition is to replace it by the condition that the alphabet "1" is always included in the solution set. Clearly, "1" covers all binary numbers and a solution set which includes this alphabet is guaranteed to cover any vector. The second condition imposes a constraint on the size of solution set. It is noted that a more accurate constraint would consider the cost of computing the alphabets in the solution set rather than its cardinality. However, this cost is annortized over M operations and is much less significant than the contribution of the additions of sub-computations. The third condition implies that all sub-computations $\alpha_k . s$ are assumed to be of equal complesity and hence, it is enough to consider the cardinality of the solution set. Note that if the size of alphabets in the solution set is not constrained, the solution set trivially consists of all the elements in V (i.e. M PRECOMPUTERS leading to fastest multiplier). If the total number of *add* operations are not constrained then the solution is trivially the set $\{1\}$ (i.e. no PRECOMPUTER leading to the slowest multiplier). The remaining two constraints are imposed to simplify the implementation of the SELECT unit. The solution alphabet set obtained for the constrained minimization problem outlined above will be referred to as the *constrained optimal solution alphabet* (COSA) set.

A. Searching the Solution Alphabet Set

We first consider the VVVS and VVFS cases. In both of these cases, the elements in the vector are not known *a priori*. Hence, a reasonable approach is to perform the minimization over the ensemble of all possible data values for a given *word-length*, W. We use an exhaustive search to find the solution alphabet set. For a given alphabet set size, Q, and a given maximum alphabet length, L, each possible combination of alphabets is generated and the corresponding average number of *add* operations per input value is computed by considering greedy decomposition for each value in the range $[0,2^{W-1}]$. Note that the results are identical for both $GD_{L\rightarrow M}$ and $GD_{M\rightarrow L}$ rules since we are considering decomposition of all possible values for a given word-length. Finally, we select the set which results in minimum number of average *add* operations per input value.

Selection of Q dictates the size (and complexity) of the MUX. If 1 is a preselected alphabet, there are $\binom{2^{L-1}}{Q-1}$ possible combinations of remaining $Q-1$ alphabets from all possible sets of length L – 1 (note that includiiig the alphabet 1, the length becomes L). L determines the complexity of the CONTROL in the SELECT unit as it determines the maximum number of bits which must be matched in alphabet identification. As this method uses exhaustive search, it gives the optimal solution to the problem of minimization of the number of average *add* operations per input value for an alphabet set of size Q with maximum alphabet length L such that 1 is also in the solution set. The minimum number of average *add* operations per input value is the optimal solution in a statistical sense for a given adaptive filtering scenario in which all possible values in the range $[0,2^{W-1}]$ are assumed equally probable. That is, the solution is optimal if the filter coefficients are assumed to be from a uniform probability density function.

The solution for the FVVS case is simpler. In this case we only consider decomposition of the values in the given input vector and find the best alphabet set for the known vector. Similar results were obtained for both $GD_{L\rightarrow M}$ and $GD_{L\rightarrow M}$ rules. One may notice that a more elaborate computation sharing approaches can be devised for this particular case by to further exploit commonality between computations since the given vector is fixed.

B. Numerical Results on Solution Alphabet Sets

We now turn our attention to numerical results. In section V-C, we present the solution sets and resulting computational gains for VVVS and VVFS situations. Section V-D presents results these results for a number of FIR filters representing the given fixed vectors.

C. VVVS and VVFS Situations

The optimal solution sets obtained for $Q=2,3$ and 4 are $\{1,3\}$, $\{1,3,5,7\}$ and $\{1,3,5,7,9,11,13,15\}$, respectively. Same solution alphabet set was obtained for all values of L $\geq log_2(Q)+1$ for a given Q. We note that the solution alphabet set consists of the set of all odd alphabets in the range $[0,2^{Q-1}]$ independent of the value of L. Since even numbers have LSB equal to zero, the zero discarding rule

ensures that no even number can ever be in the solution alphabet set. Note that the objective function to be minimized is the number of average *add* operations (sub-computations) per input value given a fixed number of alphabets in the solution set. If an odd number comprising a few bits is chosen in the solution set, the number of sub-computations into which a given random input value is broken into, is larger. If the input value is of word-length W, the minimum *add* operations possible with a solution set having L = 2 is $W/2$ sub-computations. Whereas L = *4* can reduce the sub-computations to as small as $W/4$. Hence, this decrease in sub-computations is a linear function of L. By selecting a smaller L, we increase the maximum sub-computation linearly.

On the other hand, if L is small for an alphabet, the probability that a match occurs in the input value is larger. The probability of exactly matching a number of length L is equal to 0.5^L. Note that this function decays exponentially as L increases. Hence, fewer matches occur as L is increased. As L is increased, the exponentially decaying probability of a match to occur is far stronger than the linear increase in compression (i.e. the decrease in sub-computation), and hence, it makes intuitive sense that the odd numbers with smaller length are selected, irrespective of the maximum allowed length for the alphabets. This clearly explains the elements in the solution set. One may verify that the solution set for Q = 16 is $\{1, 3, 5, \ldots, 31\}$. This result is extremely useful as it shows the regularity of the structure of alphabet set. Hence, the structure of the CONTROL part of the SELECT unit is greatly simplified due to the regularity of the solution.

Figure 6 shows the average number of *add* operations per input value for various values of Q and W. The curves are noted to be linear in the region $W \geq 2(log_2 Q + 1)$. Note that $log_2 Q + 1$ represents the maximum number of bits in an alphabet in the solution set. For $W < 2(log_2 Q + 1)$, the decrease in average computation is not proportional due to the fact that there are not enough numbers in the range $[0, 2^{\bullet\bullet} - 1]$ for the reduction to be proportionate. In this region, the lines curve upwards indicating that the longer alphabets find less than their share of matches due to a smaller number of available numbers in the range of input values. The relative reduction in the number of average *add* operations per input value compared to usual CSA multiplication is shown in figure 7. In the usual CSA multiplication, $W - 1$ *add* operations are required in a multiplication involving words to size W. Therefore, figure *7* shows the ratio of the values of figure 6 with corresponding values of $W - 1$. Consequently, we note that the curves are smooth in the region $W \geq 2(log_2 Q + 1)$ and show smaller gains in the region $W < 2(log_2 Q + 1)$ for the reason outlined above.

It is observed from figure 7 that there is a diminishing return in the relative reduction of average number of *add* operations as Q is increased. Most gain is obtained when Q is increased from 1 to 2. It is evident from the figure that lesser gains are obtained as Q is further increased. We also note that as Q increases, the complexity of the MUX in the select unit also increases and so does the delay through the SELECT unit. Hence, an optimum value of Q exists for a given implementation scheme. Further, also note that as Q increases sufficiently for a small W, the average number of *add* operations drops to zero as all product

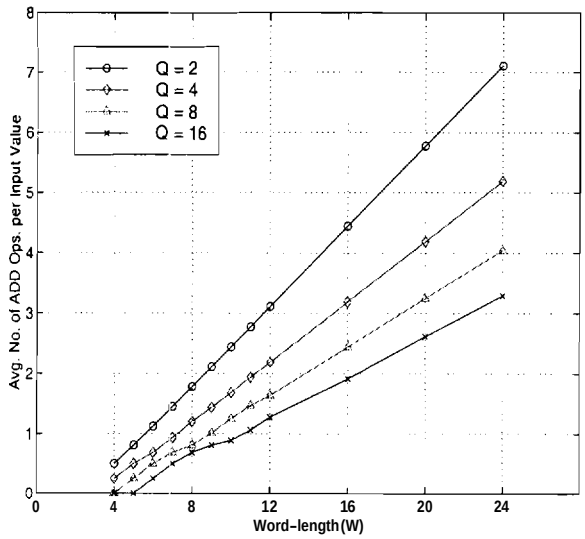

Fig. 6. Average *add* operations per input value for GD multiplier for various Q and W obtained for COSA sets.

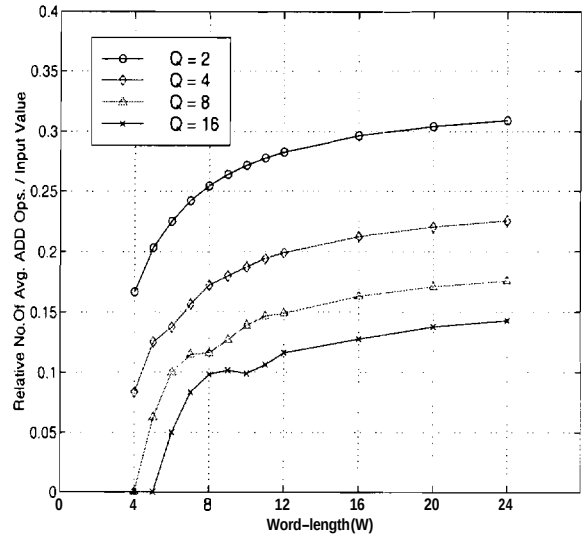

Fig. 7. Relative comparison of the average *add* operations per input value for the GD multiplier with a CSA multiplier.

values can be obtained by look-up and shift operations only. Hence, in this region the multiplier can be constructed using a single SELECT unit.

D. *FVVS Situation*

We now consider the FVVS situation. In this case, one may use the COSA sets described in section V-C (i.e. the set of odd integers $< Q$). However, since the vector is fixed and known, one can do better than the COSA set, which optimizes the average number of *add* operations in a statistical sense for input values assumed to be drawn from a uniform distribution. By average nuinber of *add* operations, we mean the average number of sub-computations per v_i required to compute the scaled vector Vs. Since, the most interesting application of FVVS situation is non-adaptive filtering, we consider several example FIR filters to obtain the relative gains in operation reduction when the best alphabet sets for the given filters are used instead of the alphabets in COSA set. These filters include elliptic, Butterworth, Parks-McClellan and least squares filters, both band-pass as well as low-pass. Hence, the examples considered span a variety of filter lengths and types. The coefficients obtained for these filters were *maximally scaled.* By maximal scaling, we mean that each coefficient, v_i, is expressed as $v_i' \times 2^{-p}$, such that $p \geq 0$ and $v_i' \geq 2^{W-2}$. Hence, the coefficient applied to the input of the inultiplier is v_i' instead of v_i, and scaling ensures that most of the MSBs are utilized in multiplication. Such scaling operations are routinely used in filter implementations in order to reduce the effects of quantizations on the filter performance [8]. Maximal scaling ensures minimal quantization error in filtering computations using fixed size integer multiplier units and coarsely mimics a floating point type of operation without incurring the complexity of implementing a floating point unit. The best alphabet sets for the example filters and the effect of M, Q, L and W on the average reduction

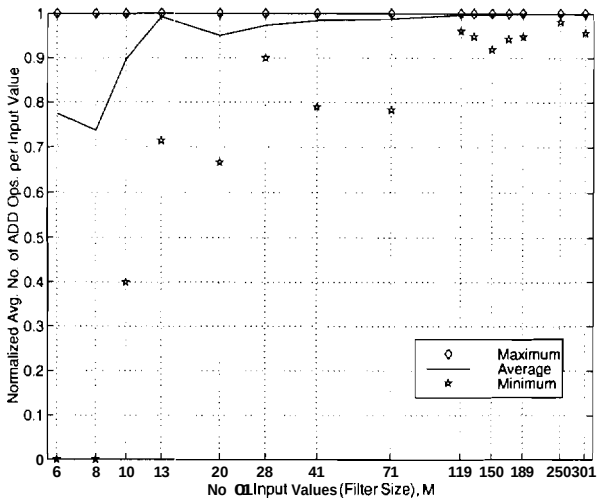

Fig. 8. Average add operation per FVVS input normalized by the corresponding value obtained for the COSA set for various example filters. W, Q and L lumped, M shown on abscissa.

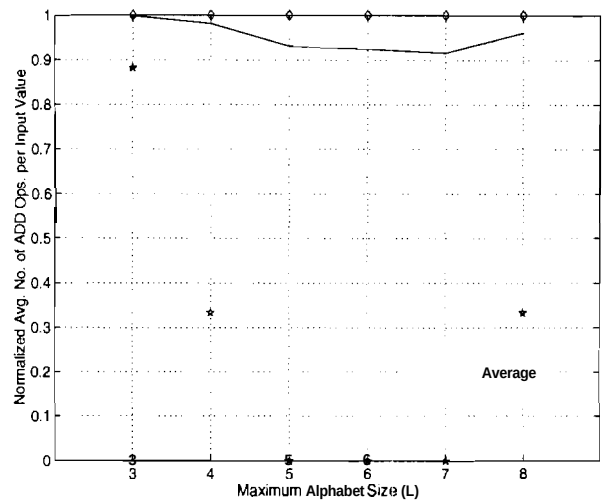

Fig. 9. Average add operation per FVVS input normalized by the corresponding value obtained for the COSA set for various example filters. M, W and Q lumped, L shown on abscissa.

in computation is shown in figures 8 — 11. These results compare the optimal alphabet sets for the given filter with the COSA sets for various Q (recall that COSA set depends on Q only.). Since the number of parameters varied is large, the results are most effectively summarized by showing the minimum, average and maximum relative values of the average add operations per input value with respect to COSA set averages. In figure 8, the filter taps are on the abscissa, and the relative gain in average computation over COSA set is shown on the ordinate. Clearly, the gains decrease as M increases. For :;mall filter sizes, relatively large gains are obtained because appropriate choice of alphabets can reduce the computation to zero! In this case, the multiplier can be constructed by using a single SELECT unit. As M increases, the tap values start to mimic numbers drawn from a uniform distribution of random numbers and the relative gains over COSA sets diminishes. This is clearly exhibited by the curve displaying the average value of the relative gain. We also found that for most examples, COSA sets were the optimal choice even for non-adaptive filters. The results indicate that on an average, for small filter sizes, the average computation per multiplication improves by a factor of 0.8 over the COSA set averages

Figure 9 shows the results with L on the abscissa. As L is increased, smaller filters can be implemented by a single SELECT unit. However, for most filter examples, the relative gains over COSA set averages are just nominal as a function of increasing L. Figure 10 shows the effect of varying W on the relative reduction in average computation over the COSA set averages. Most gains are obtained for smaller W. This figure shows that zero computation filters are filters implemented with $W = 8$ only. In these filters, M = 6 and 8, as evident from figure 8, and L = 5, 6 and 7, as shown by figure 9. Further, by table 11, these filters have $Q = 4$ and 8. Hence, it is quite clear that most gain in computation reduction over the COSA set average computation are obtained for small filters and small word-lengths. As the filter size

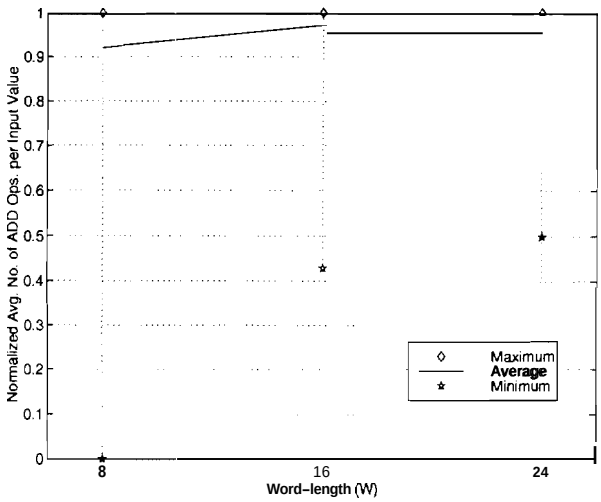

Fig. 10. Average add operation per FVVS input normalized by the corresponding value obtained for the COSP. set for various example filters. M, L and Q lumped, W shown on abscissa.

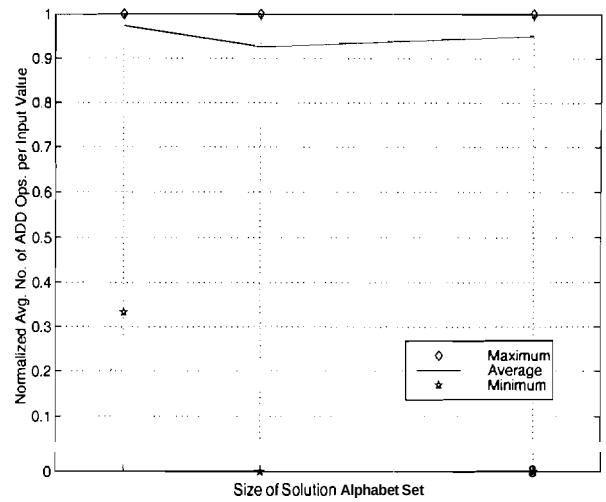

Fig. **11.** Average add operation per FVVS input normalized by the corresponding value obtained for the COSA set for various example filters. M, L and W lumped, Q shown on abscissa.

increases, and/or W increases, the gains become smaller and COSA set yields close to optimum sharing even for non-adaptive filters.

The reader is reminded that the results presented in this section consider maximally scaled filters which is the worst case situation. If the coefficients were *unscaled*, most of the bits in their MSB part would be zeros and significantly lesser computation would be required even for longer filters. This is because most non-adaptive filters try to implement a "brick-wall" transfer function, and their values lie along a $sinx/x$ function. Hence, the MSB bits rapidly become zeros if they are expressed in sign-magnitude form. The maximally scaled coefficients provide the *worst case* scenario for the presented computation sharing technique, since the entire range of W bits is used once the coefficient is scaled. For any scaling less than maximal, the gains over usual CSA multiplication would be considerably larger for both COSA set, as well as the optimal solution set for FVVS case. Hence, significantly less sub-computations would be required over conventional multiplication scheme, thereby resulting in faster and more efficient filter implementation.

VI. Computation Sharing Multiplication Structures

A. GD Multiplication

Figure 12 shows the structure of a multiplier using the COSA set. This example shows an 8-bit multiplier and the numbers in parenthesis along the lines indicate the width of the buses. The GD rule requires discarding any number of leading zeros preceding an alphabet. We will refer to the alphabet separating sequence of zeros as *trailing zeros* for $GD_{L\rightarrow M}$ and *leading zeros* in the case of $GD_{M\rightarrow L}$. Each alphabet is identified and extracted, and the corresponding sub-computation added to the product. Since we do

not know where the boundary of the next alphabet begins, alphabet extraction can only be done in a sequential manner and this approach leads to a sequential multiplier. This follows from the fact that with the GD rule, parallel identification of alphabets is not easy to implement due to the variable nature of the alphabet boundaries,

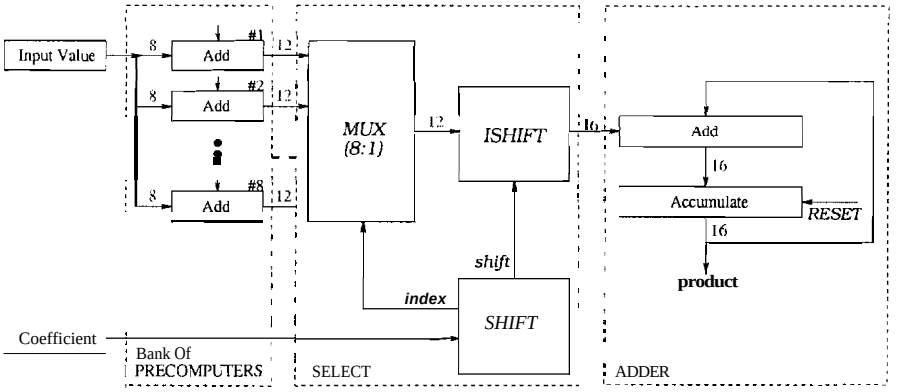

Fig. 12. Sequential (SAA) multiplication scheme for Q = 8.

Consider a $GD_{L\rightarrow M}$ multiplier first. The goal of the CONTROL unit is to discard the trailing zeros at the end of every alphabet in the input value and to identify the longest matching odd number in the range $[0, Q]$. The zeros are discarded by a SHIFT unit which continues to shift the input value as long as the leading bit is not a 1. The final output of the shifter is an odd number such that the $log_2 Q + 1$ LSBs represent the desired alphabet. This alphabet can be then used to provide the select signal to the MUX. Note that the alphabet is always odd, whereas the MUX select lines take all values in the range $[0, log_2 Q]$. A simple scheme is to use discard the LSB and use the next $log_2 Q$ lines as direct inputs to the MUX select lines. This is equivalent to ***divide-by-2*** operation which converts the alphabet to the ***index*** of the appropriate look-up entry (i.e. pre-computed sub-computation). Hence, the CONTROL unit is no more complex than a simple arithmetic shifter. The amount by which the input value is right shifted is compensated for by the ISHIFT (Inverse SHIFT) unit which uses the same control signals to provide an identical shift in the opposite direction. After extracting the first alphabet, the SHIFT unit removes $log_2 Q + 1$ bits from the input value. The remaining value with these bits removed can either be fed-back to the input of the same SHIFT unit, or, to another SHIFT unit. The correct value of total number of bits removed is provided to the ISHIFT unit so as to ensure the correctness of the operation. This requires keeping a track of the total number of bits processed so far before processing the next alphabet.

The $GD_{M\rightarrow L}$ multiplier simplifies keeping track of the count of the bits processed and offers a better alternative. The input value is scanned from MSB to LSB and leading zeros before the first alphabet are removed. The next $log_2 Q + 1$ bits are used to select the correct pre-computed sub-computation. Recall that in our convention, the COSA alphabets are identical in $GD_{L\rightarrow M}$ and $GD_{M\rightarrow L}$ since bit reversals are assumed implicit. Hence, COSA alphabets $\{1, 3, 5, 7\}$ in $GD_{M\rightarrow L}$ mean the alphabets $\{4, 6, 5, 7\}$ as the

scan direction is reversed. Consequently, 1 and 3 in $GD_{M\rightarrow L}$ for the Q = 4 alphabet set are implemented by 4 and 6, respectively. The output of the MUX unit are connected to the rnost significant bits of the *add* unit. The ISHIFT unit reverses the shift operation done by SHIFT unit by performing a right shift of the partial accumulated result by the number of zeros discarded. The partial result always fills the MSB side of the product bits, and, subsequent operations move bits towards right, aligning the partial result with the next sub-computation. In this configuration, the ISHIFT unit appears at the output of the ACCUMULATE unit and subsequent sub-computations are aligned by right-shift operations. As the product bits are computed, each subsequent sub-coinputation shifts the product bits in the ACCUMULATE unit by $log_2Q - 1 + z_j$ bits, where z_j is the number of leading zeros in jth sub-computation. The amount $log_2Q - 1$ can be added to z_j by re-assigning the control lines such that the value z_j on shift control lines results in a shift by $log_2Q - 1 + z_j$ bits.

The relative speed advantage of the GD multiplier over a carry-save array multiplier will be called *delay efficiency.* For the $GD_{L\rightarrow M}$ multiplier, it is given as $\eta_{GD,L\rightarrow M} = \tau_{CSA}/\tau_{GD,L\rightarrow M}$ where τ_{CSA} and $\tau_{GD,L\rightarrow M}$ are delays through the CSA and $GD_{L\rightarrow M}$ multipliers, respectively. The delay efficiency for the $GD_{M\rightarrow L}$ multiplier is similarly defined. The delay efficiency is easily computed by accounting for the driver fan-outs, loading effects for the given SELECT unit structure. We considered pass-gate based SHIFT, MUX and ISHIFT units and considered the loading on appropriate drivers to obtain the delay efficiency curves shown in figure 13. The figure shows the delay efficiency obtained for the GD multiplier assuming the following normalized parameter values: $\tau_{INV} = \tau_{pg} = 1$, $\tau_{nand} = 1.5$, $\tau_{HA} = 2$ and $\tau_{FA} = 3$, where $\tau_{INV}, \tau_{pg}, \tau_{nand}, \tau_{HA} and \tau_{FA}$ represent the delay through an inverter, pass-gate, nand gate, half-adder and full-adders, respectively. We observe that most computational advantage due to sharing is lost in the sequential scheme due to the recurring τ_{SEL} in the critical path for every sub-computation. Hence, the alphabet search must be performed in a parallel fashion such that the search does not severely degrade the computational advantage. Figure 13 shows that despite the recurring τ_{SEL} delay, speed advantage of about 25% can be achieved for the single-stage SHIFT unit. Further, the delay efficiency decreases as W increases due to increased drive requirements. In the case of logarithmic shifter, the overheads and the recurring τ_{SEL} leads to a delay efficiency which is less than 1! Clearly, we need to investigate alternative structures which modify the decoding rule such that alphabets can be searched in parallel. The simplest way to achieve that is to remove the condition of variable alphabet boundaries. This is explored in the next section.

B. Fixed Size Look-up Multiplication

The major draw-back of the GD multiplication is the variability of alphabet partitions. Consequently, it is not possible to construct a parallel multiplier to speed up computation. If the alphabet boundaries can be fixed, one can use parallel SELECT units to search for matching alphabets starting from known positions, thereby providing the option of simpler multiplier. Hence, alternative decomposition rule is

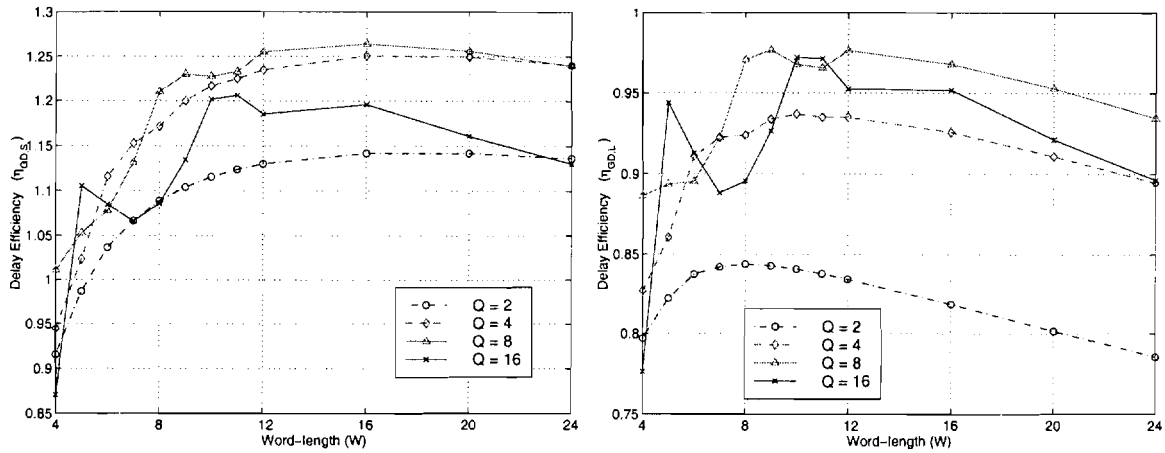

Fig. 13. GD multiplier delay efficiency for various W and Q with (a) Single-stage SHIFT unit, **(b)** Logarithmic SHIFT unit.

needed to enable parallel search for alphabets in the input value.

One such rule is a *fixed size look-up* (FSL) rule in which we only discard a fixed maximum number of trailing zeros. Each SELECT unit considers a maximum of $L = log_2 Q + 1$ bits in identification of the alphabet. Hence, the input value is decomposed into $\lceil W/L \rceil$ parts, each of which is processed independently. In a parallel multiplication scheme, each of these $\lceil W/L \rceil$ parts is applied to a separate SELECT unit as shown in figure *14*. The SHIFT unit must only provide a maximum shift of $L - 1$ bits since at most: we could encounter $L - 1$ leading zeros before the first alphabet. The leading zeros are simply discarded and the resulting right-shifted number is used to generate the *index* value for the sub-computation, similar to the GD multiplier, by shifting the number once to the right. The ISHIFT simply inverses the operation performed by the SHIFT by providing an opposite shift of *exactly* the same amount using the same shift control values. Its complexity is identical to the complexity of the SHIFT unit.

The upper SELECT unit generates the product of L LSB bits of the input value with the scalar, s. The lower SELECT unit produces the product of next L bits with s. Hence, a shift of L bits is provided using interconnect wiring when feeding these two sub-computations to the ADDER. This shift operation does not increase any computation overhead. This scheme require $\lceil W/L \rceil - 1$ *add* operations independent of the input value. In comparison, the usual parallel multiplication in an array fashion requires $W - 1$ computations. Hence, the FSL rule provides a reduction in average *add* operations per input value by a factor of $(\lceil W/L \rceil - 1)/(W - 1)$ which is its delay efficiency. The example in figure *14* considers both W and Q equal to 8. Hence, $L = 4$ and $W/L - 1 = 1$ *add* operations are needed. This scheme reduces the average computation by a factor of 7. We note from figure 6 that this case requires no *add* computation in the GD multiplier and implements multiplication through the use of a single SELECT unit.

The advantage of removing recurring τ_{SEL} delay in the sub-computations is clearly exhibited in figure 15. which show FSL multipliers constructed using a single stage and $log_2 Q$ stage SHIFT units, respectively.

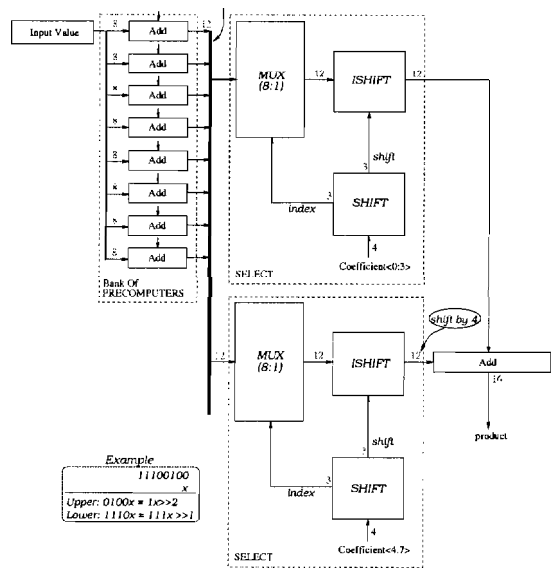

Fig. 14. Parallel 8-bit multiplication scheme using FSL rule for $Q = 8$.

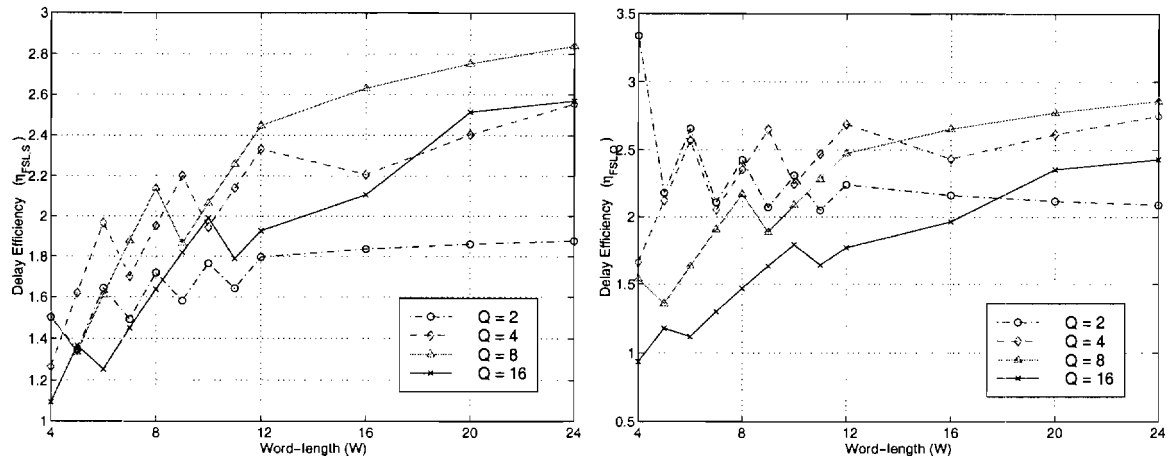

Fig. 15. FSL multiplier delay efficiency for various W and Q with (a) Single-stage SELECT unit, (b) $log_2 Q$-stage SELECT unit.

The results clearly indicate that **2-3** times computation speed-up is easily achievable using the FSL scheme. Further gains may be achieved by considering more elaborate methods of parallel alphabet search.

A close observation of the array type FSL multiplier shows that the maximum of the delays through the first two SELECT units occur on the critical path. Hence, it must be kept small. As we progress down the array, the latency available to the SELECT units increases and one may use a higher Q SELECT unit in subsequent subcomputations. This is shown in figure 16 which shows a 14 x 14 in (a) and a 16×16 multiplier in (b), constructed using SELECT units of increasing alphabet size. Clearly, if the delay through the third SELECT unit can be made equal to the sum of the delays through the first SELECT unit and the half-adder, one can significantly reduce the computation further. This technique of

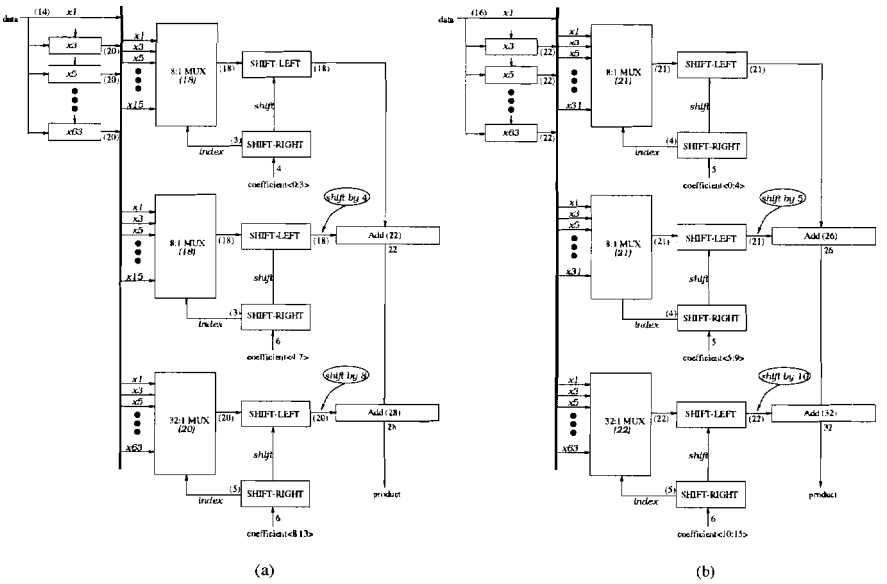

Fig. 16. Possible configurations for (a) 14 × 14, and (b) 16 × 16 multipliers using parallel FSL multiplication schemes.

balancing the path delays using *tapering* of SELECT units is a well known approach used in logic synthesis. The ideal ccmputation gains shown in figure 7 ignore τ_{SEL} which increases with increasing values of Q and W. However,.they can provide a rough base-line for evaluating the performance of a decoding rule and its implementation.

In order to verify the computational advantage of the proposed FSL multiplication scheme, we implemented an 8×8 bit array multiplier using an alphabet set size $Q = 8$ and $L = 4$ in 0.6μ CMOS technology. A 3-stage SHIFT unit was employed in the multiplier. We also implemented a CS array multiplier and compared the delay and power of the two schemes using analog simulation. The proposed multiplier had a delay efficiency of 1.85 and it consumed 1.9 times more power than the CSA multiplier. Using voltage scaling, one could reduce the power dissipation of the proposed multiplier by a factor of approximately 0.29, thereby, resulting in a power advantage over the normal CSA multiplier by 44%. We observe that these gains are mainly dependent on the choice of Q and L and the delay $\tau_{SEL,Q}$ and various trade-offs exist as these selections are changed. Hence, a carefully designed SELECT unit greatly improves the relative advantage of the proposed scheme.

The computational advantage of using the GD multiplier as shown in figure 7 ignores the delay through the SELECT unit as the lookup of pre-computed values are riot considered as "computations". The actual advantage of the proposed computation sharing multiplication is dependent on implementation as well as the circuit style used. The multiplication scheme presented here must deal with relatively larger fan-outs and a technology which is relatively insensitive to larger fanout will yield gains that are closer to the ones shown in figure 7. A close observation of the units shown in the example implementations shown in this paper shows that these implenlentations favor domino CMOS circuit style. It is further

noted that the FSL multiplier can also be pipelined. Hence, the computational advantage of using a computation. sharing multiplication can be further improved by considering alternative circuit styles and technologies Additionally, the proposed schemes also give an opportunity to explore further computation sharing by considering more intricate inter-relationships between the elements of given vectors by adding a small storage overhead. Hence, the proposed schemes offer more flexibility in reducing computational redundancy in a given DSP algorithm implementation.

VII. Conclusion

We present a multiplication approach which specifically targets reduction of redundant computation in common *digital signal processing* (DSP) tasks such as filtering and matrix multiplication. We showed that such tasks can be expressed as multiplication of vectors by scalars and by sharing computation in such *vector scaling* operations, the execution time of such tasks can be reduced. We presented computation sharing multiplication schemes which considerably reduce redundant computation by decomposing the vectors in a manner which results in maximal computation sharing, thereby, resulting in a faster and potentially low-power implementation. Two decomposition approaches were presented. First, based on a *greedy decomposition,* and, the second, based on a *fixed-size lookup* rule. It was shown that the delay efficiency of the proposed multipliers is only limited by the delay through the SELECT unit. Analysis of the proposed implementations based on FSL rule show the speed-up by a factor of up to **3.** This is confirmed by analog simulation of an example multiplier which yields a speed advantage by a factor of about 1.85 over a conventional carry save array multiplier and a power disadvantage of 1.9. Using voltage scaling, we could reduce the power dissipation by a factor of 0.29, thereby, obtaining 44% reduction in power over a CSA multiplier. Hence, the ideas presented in this paper can assist design of DSP algorithms and their implementations for high-speed applications. Alternatively, using voltage scaling, one can significantly reduce the power consumption of such applications for any desired performance.

References

[1] K. Muhammad and K. Roy, "Very Low-Complexity Digital Filters Based On Computational Redundancy Reduction," *Submitted to IEEE Transactions on VLSI systems.*

[2] J. M. Rabaey, "Digital Integrated Circuits: A Design Perspective," Prentice Hall, New Jersey, 1996.

[3] N. H. E. Weste and K. Eshraghian, "Principles of CMOS VLSI Design: *A* Systems Perspective," *2nd Edition,* Addison Wesley, 1994.

[4] E. E. Swartzlander, "Computer Arithmetic," *IEEE Computer Society Press,* 1990.

[5] S. E. McQuillan and J. V. McCanny, "A Systematic Methodology for the Design of High Performance Recursive Digital Filters," *IEEE Trans. on Computers,* Vol. 44, No. 8, pp. 971-982, Aug. 1995.

[6] N. Sankarayya, K. Roy, and D. Bhattacharya, "Algorithms for low power and high speed FIR filter realization using differential coefficients," *IEEE Trans. Circuits and Systems,* Vol. 44, No. 6, pp. 488-497, Jun. 1997.

[7] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," The MIT Press, 1990.

[8] E. Cooper, "Minimizing Quantization Effects Using the TMS320 Disgital Signal Processor Family:" *Application Report, http://www.ti.com/sc/docs/psheets/abstract/apps/spra035.htm,* Texas Instruments, 1994.