ABSTRACT

This paper presents an efficient hardware implementation of spatial domain two-dimensional IIR and FIR filters. We use a state space representation to achieve efficient algorithms and then use a common computational primitive to optimize the hardware implementation. We present a multiprocessor implementation for a real-time or near real-time two-dimensional linear system with a raster scan input. The multiprocessor system has a simple control scheme, a simple interconnection network, a very high efficiency, and low data transfers.

INTRODUCTION

Two-dimensional (2-D) spatial domain filters are used in many applications with image processing being the most common. In addition, the 2-D spatial domain digital filter can be generalized to include all discrete, linear shift-invariant (DLSI) systems. This means that many common 2-D digital systems can be represented as spatial domain filters. The only requirement is for the system to be represented by a difference equation with constant coefficients.

In this paper, we summarize previous results on the implementation of 2-D IIR digital filters and present a new algorithm for implementing 2-D FIR filters using the same computational primitive as was previously derived for 2-D IIR filters [1]. We present a real-time 2-D IIR and FIR filter implementation based upon using a common computational primitive. The computational primitive is based upon the state space representation. The computational primitive is based upon the state space representation. Using this approach, every state variables and output can be computed by a simple computational primitive [2]. The computational primitive for a 2-D IIR filter requires 2 multiplications and 2 additions. Using the same approach, the computational primitive for a 2-D FIR filter requires one multiplication and two additions. In this paper, we develop a new computational primitive for the 2-D FIR filter which also requires 2 multiplications and 3 additions. Thus, we can develop a single processor which can efficiently implement both 2-D FIR filters and 2-D IIR filters using the same computational primitive. This improves the efficiency of the hardware realization and makes it more versatile.

We previously developed a prototype DSP chip [3] which will be used for the our prototype system to implement 2-D IIR filter up to fourth order in real-time. The processor was designed based upon the computational primitive for the 2-D IIR filter. In this paper, we present a modified processor design which can accommodate 2-D FIR filters more efficiently without sacrificing performance for the 2-D IIR filter.

The Efficient Real-Time Spatial Domain 2-D IIR And FIR Digital Filter Implementation

Jae G. Jeong
Rensselaer Polytechnic Institute
Electrical and Computer Engineering
Rensselaer, New York

ABSTRACT

This paper presents an efficient hardware implementation of spatial domain two-dimensional IIR and FIR filters. We use a state space representation to achieve efficient algorithms and then use a common computational primitive to optimize the hardware implementation. We present a multiprocessor implementation for a real-time or near real-time two-dimensional linear system with a raster scan input. The multiprocessor system has a simple control scheme, a simple interconnection network, a very high efficiency, and low data transfers.

INTRODUCTION

Two-dimensional (2-D) spatial domain filters are used in many applications with image processing being the most common. In addition, the 2-D spatial domain digital filter can be generalized to include all discrete, linear shift-invariant (DLSI) systems. This means that many common 2-D digital systems can be represented as spatial domain filters. The only requirement is for the system to be represented by a difference equation with constant coefficients.

In this paper, we summarize previous results on the implementation of 2-D IIR digital filters and present a new algorithm for implementing 2-D FIR filters using the same computational primitive as was previously derived for 2-D IIR filters [1]. We present a real-time 2-D IIR and FIR filter implementation based upon using a common computational primitive. The computational primitive is based upon the state space representation. The computational primitive is based upon the state space representation. Using this approach, every state variables and output can be computed by a simple computational primitive [2]. The computational primitive for a 2-D IIR filter requires 2 multiplications and 2 additions. Using the same approach, the computational primitive for a 2-D FIR filter requires one multiplication and two additions. In this paper, we develop a new computational primitive for the 2-D FIR filter which also requires 2 multiplications and 3 additions. Thus, we can develop a single processor which can efficiently implement both 2-D FIR filters and 2-D IIR filters using the same computational primitive. This improves the efficiency of the hardware realization and makes it more versatile.

We previously developed a prototype DSP chip [3] which will be used for the our prototype system to implement 2-D IIR filter up to fourth order in real-time. The processor was designed based upon the computational primitive for the 2-D IIR filter. In this paper, we present a modified processor design which can accommodate 2-D FIR filters more efficiently without sacrificing performance for the 2-D IIR filter.

The key to success of any multiprocessor architecture is the partitioning of algorithms so each processor is effectively used. We have used a state space representation as a vehicle to aid in the partitioning of 2-D DLSI systems for implementation on a multiprocessor system. The state space representation provides the potential for minimizing the data communication requirements for a given algorithm without increasing computational complexity.

ALGORITHM

A 2-D digital filter can be modeled as a discrete linear shift-invariant (DLSI) system and can be implemented efficiently using the state space approach. A general order 2-D DLSI system with quarter plate support can be represented by the finite difference equation as is given by

\[
g(m, n) = \sum_{i=0}^{M} \sum_{j=0}^{N} a(i, j) f(m-i, n-j) - \sum_{i=0}^{M} \sum_{j=0}^{N} b(i, j) g(m-i, n-j)
\]

(1)

The parameters \(a(i, j)\) and \(b(i, j)\) are coefficients which determine the characteristics of the system. If all of the \(b(i, j)\) are equal to zero, then we have an FIR filter. Otherwise, eqn. (1) represents a IIR filter. By taking the z-transform of eqns. (1), we can write the relationship between the transform of the input sequence, \(F(z_1, z_2)\) and the transform of the output sequence, \(G(z_1, z_2)\) as follows:

\[
G(z_1, z_2) = a(0, 0)F(z_1, z_2) + \sum_{i=0}^{M} \sum_{j=0}^{N} [a(i, j) F(z_1, z_2) - b(i, j) G(z_1, z_2)] z_1^i z_2^j
\]

(2)

From this equation, we can obtain the data flow graph of the 2-D DLSI system shown in Figure 1. In this figure, there are two kinds of delay elements. \(z_1^{-1}\) is the pixel delay and \(z_2^{-1}\) is the row delay. We assigned the horizontal state variable \(r\) in as the input to each \(z_1^{-1}\) delay and the vertical state variable \(q\) in as the input to each \(z_2^{-1}\) delay. From this graph, we can obtain the following state space equations:

\[
g(m, n) = c_0 f(m, n) + g(m, n)
\]

(3)

\[
r_k(m, n) = c_{mk} f(m, n) + d_{mk} g(m, n) + r_{k-1}(m-1, n)
\]

(4)

\[
q_k(m, n) = c_{mk} f(m, n) + d_{mk} g(m, n) + r_{M(N+1-k)-1}(m-1, n) + q_{k-1}(m, n-1)
\]

(5)

Where
Figure 1: A data flow graph for the general order 2-D DLSI system

The modified coefficients c's and d's are obtained from the a's and b's and y is a temporary variable defined for simplification.

The equations, eqn. (3) to eqn. (5), are used to design the system. Among them, eqn. (5) is the most complex. If we build a processor which can compute this equation in a single cycle, we can compute each equation from eqn. (3) to eqn. (5) in a single cycle. Therefore, we define eqn. (5) as a computational primitive for the processor. It requires two multiplications and three additions.

For a 2-D FIR filter, eqn. (3) to eqn. (5) can be modified by setting the d_{a,k} and d_{b,k} to zero as follows:

\[
g(m,n) = c_0 f(m,n) + r_{M-1}(m-1,n) + q_{N-1}(m,n-1)
\]

\[
r_a(m,n) = c_{a,k} f(m,n) + r_{k-1}(m-1,n) + q_{k-1}(m,n-1)
\]

\[
q_a(m,n) = c_{q,k} f(m,n) + r_{M(N+1-k)-1}(m-1,n) + q_{k-1}(m,n-1)
\]

Where

\[
c_0 = q(0,0)
\]

\[
c_{a,k} = a(M - i,j)
\]

\[
c_{q,k} = q(0, N - j)
\]

\[
r_a(m,n) = 0 \quad \text{for } k < 0 \text{ or } n < 0
\]

\[
q_a(m,n) = 0 \quad \text{for } k < 0 \text{ or } n < 0
\]

From the above equations, we define a computational primitive for the 2-D FIR filter which requires one multiplication and two additions.

THE COMMON COMPUTATIONAL PRIMITIVE

If we implement the processor based on one specific computational primitive, the other algorithm, which has a different computational requirement, can not be implemented efficiently with this processor. We can achieve higher efficiency by finding a common computational primitive for both 2-D FIR and IIR filters.

There are two approaches to obtain the common computational primitive. The one, which is commonly used, is to implement the simpler computational primitive. Then, we can implement the complex one by using the simpler one several times. This approach will result in low throughput or more complex system hardware. The other approach is to implement the complex computational primitive. Then, we can implement the algorithm which requires many simple computations more efficiently by changing the simple computations into the more complex computational primitive. We choose this latter approach because we discovered that we can use the block state implementation to change the computational primitive for the 2-D FIR filter to be the same as that for the 2-D IIR filter. This approach has many benefits such as reducing the number of computational cycles required for each algorithm and reducing the number of processing elements required for a real-time implementation. However, it does increase the complexity of the processor.

Today's VLSI technology is developing very rapidly. In a near future, we can implement more complex circuits in a single DSP chip. There are two ways to utilize this development. One way is to use more complex processor implementations, and the other is to use more processing elements on a single chip. The effectiveness of each approach depends on the algorithm.

IMPLEMENTATION WITH THE COMPUTATIONAL PRIMITIVE FOR THE 2-D FIR FILTER

The implementation of the computational primitive for the 2-D FIR filter is straightforward as shown in figure 2. This figure shows the computational primitive and the arithmetic unit (AU) of the processor for the 2-D FIR filter. The processor was implemented with three pipeline stages so that it computes one computational primitive for the 2-D FIR filter in a single cycle after pipelines are filled.

The implementation of the computational primitive for the 2-D IIR filter can be implemented easily in 2 cycles with a small change in the AU circuit. The only change required is a feedback loop for adder \#2 in the Figure 2.
IMPLEMENTATION WITH THE COMPUTATIONAL PRIMITIVE FOR THE 2-D IIR FILTER

In this section, we present the implementation based on the computational primitive for the 2-D IIR filter. Figure 3 shows the computational primitive and AU of the processor for the 2-D IIR filter. This processor can compute one computational primitive for the 2-D IIR filter in every 2 cycles after pipeline stages are filled.

As you can see in the table, the results of the computation for the computational primitive are obtained at the end of cycle 3, cycle 5, and so on. Therefore, this processor can compute one computational primitive for the 2-D IIR filter in every 2 cycles after pipeline stages are filled.

<table>
<thead>
<tr>
<th>cycle</th>
<th>multiplier stage #1</th>
<th>multiplier stage #2</th>
<th>adder #1</th>
<th>adder #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c * f</td>
<td>d * y</td>
<td>r + q</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>c * f</td>
<td>d * y</td>
<td>r + q</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c * f</td>
<td>d * y</td>
<td>r + q</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d * y</td>
<td>c * f</td>
<td>d * y</td>
<td>r + q</td>
</tr>
<tr>
<td>4</td>
<td>c * f</td>
<td>d * y</td>
<td>r + q</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>d * y</td>
<td>c * f</td>
<td>c * f</td>
<td></td>
</tr>
</tbody>
</table>

As you can see in the table, the results of the computation for the computational primitive are obtained at the end of cycle 3, cycle 5, and so on. Therefore, this processor can compute one computational primitive for the 2-D IIR filter in every 2 cycles after pipeline stages are filled.

The derivation of the block state space equation is given in several references [4, 5]. We only emphasize the derivation of the block state space representation of the 2-D FIR filter with the block size of 2 \times 1 to simplify our discussion. We divide the input and output sequences into blocks of length 2, and define the input and output data as streams of vectors of length 2 [5].

The scalar state space equations are shown in eqn. (6) to eqn. (8). From these scalar state space equations, we can derive the block state space equations by applying the original state space equation recursively. The following equations are obtained from the eqn. (7):

\[
\begin{align*}
r_h(m, n) &= c_{0,h}(m, n) + r_{k-1}(m - 1, n) \\
r_h(m - 1, n) &= c_{0,h}(m - 1, n) + r_{k-1}(m - 2, n) \\
g(m, n) &= c_{0,f}(m, n) + c_{M,1}(m - 2, n) + g_{k-1}(m, n - 1) \\
g(m - 1, n) &= c_{0,f}(m - 1, n) + r_{k-1}(m - 2, n) + g_{k-1}(m - 1, n - 1) \\
q_0(m, n) &= c_{0,b}(m, n) + c_{s,M}(n + 1, n) + f(m - 1, n) + r_{M}(n + 1, n) + q_{k-1}(m, n - 1)
\end{align*}
\]
\[ q_k(m-1,n) = c_{a,k}f(m-1,n) + r_{M(N+1-k)}(m-2,n) + q_{a-1}(m-1,n-1) \]  

(15)

where

- \( c_a \) = \( q(0,0) \)
- \( c_{a,k} \) = \( q(M - i,j) \) for \( 0 \leq i \leq M - 1, 0 \leq j \leq N \) and \( k = i + jM \)
- \( c_{b,k} \) = \( q(0,N - j) \) for \( 1 \leq j \leq N \) and \( k = j \)
- \( r_k(m,n) = 0 \) for \( k < 0, m < 0, \) or \( n < 0 \)
- \( q_k(m,n) = 0 \) for \( k < 0, m < 0, \) or \( n < 0 \)

With this modification, all the computations can be mapped into two multiplications and three additions which is the same as the computational primitive for the 2-D IIR filter.

For the scalar state space implementation of a 2-D FIR filter, we need \( n(n + 1) \) horizontal state variable computations, \( n \) vertical state variable output computations, and one output computation. A total of \( n^2 + 2n + 1 \) computations are required to obtain one output. Therefore, the throughput with single processor is \( 1/(n^2 + 2n + 1) \) outputs per cycle for the scalar state space implementation and \( 2/(n^2 + 3n + 2) \) outputs per cycle for the block state space implementation with the block size of \( 2 \times 2 \).

Thus, the throughput with the block state space implementation is improved by almost a factor of two.

**MULTIPROCESSOR IMPLEMENTATION**

The overall implementation is shown in figure 4. In this implementation, we define one row of data as one data block; we assign the first row of data to the first processor, the second row of data to the second processor, and so on. Generally, the number of processors is much smaller than the number of rows. If the number of processor is four, then we assign the fifth row of data to the first processor, the sixth row of data to the second processor, and so on. With this implementation scheme, we can achieve the following advantages:

- **Reduced amount of data transfers:** According to the algorithm (see eqn. (4) and eqn. (15)), the horizontal state variable, \( r \), has to be transferred from the processor which processes the \((m-1,n)\) pixel to the processor which processes the \((m,n)\) pixel. Since both of these pixels are processed by the same processor according to our implementation scheme, the horizontal state variables do not need to be transferred to any other processors.

- **Local data transfers:** According to the algorithm (see eqn. (5)), the vertical state variable, \( q \), has to be transferred from the processor which processes the \((m-1,n)\) pixel to the processor which processes the \((m,n)\) pixel. According to our implementation scheme, these two processors are the nearest neighbors. Therefore, only local data transfers are required.

The use of this scheme permits us to achieve real-time throughput with a given number of processors. Each processor, as shown in figure 4, processes one row of data (one block of data). The timing diagram shown in figure 5 explains how we can achieve real-time throughput with multiple processors with this implementation scheme. This figure was based on the assumption that the input data interval is \( 127 \) nsec. This rate is equivalent to the 30 frames per second of \( 512 \times 512 \) image. We also assumed that the processor can compute one computation primitive in 100 nsec, and that four equations need to be computed.

As you can see in the figure, processor number 1 starts processing as soon as the row 1 of data is available. During the processing, the processor computes the vertical state variables, \( q \). They will be used for the processing of the next row of data. When the row 2 of data is available, processor number 2 can start the processing because the required vertical state variables are already available. Processor number 1 can start the processing of row 5 of data as soon as the input data is available because the resource (processor) and all the required data (input data and vertical state variables) are available.

After processor number 4 starts the processing, every processor is simultaneously computing as long as input data is available. Therefore, we can achieve real-time processing with four processors.

All the processors are connected in a linear array with data communications only with the input control block, the output control block, and the nearest neighbors. Data communications between processors are always first-in first-out and buffered. If data is available, the processor operates at its full speed. Since the processors operate asynchronously, speed up is essentially linear as we add additional processors.

For the general \( n \)-th order system based on the scalar state space implementation, we need \( n^2 + 2n + 1 \) computations for each input data. Therefore, we can achieve real-time throughput with \( n^2 + 2n + 1 \) processors if each processor can compute one computational primitive in a single cycle. The number of
processors can be greatly reduced by using the block state space implementation. The required number of processors for the \( n \)-th order system becomes \((n^2 + 3n + 2)/2\). Thus, the number of processors is reduced by almost one half for the high order system.

Generally, the FIR filter requires much higher order implementation than the IIR filter. Therefore, this block state space implementation reduces the system size greatly.

**PERFORMANCE EVALUATION**

Let \( T_I \) be the number of cycles to compute eqn. (1) with a single processor which can compute one multiplication or one addition in a single cycle. Direct implementation of the 2-D IIR filter requires \( 2(n + 1)^2 - 1 \) multiplications and \( 2(n + 1)^2 - 2 \) additions. Thus, \( T_I \) is equal to \( 4(n + 1)^2 - 3 \). Let \( P \) be the number of functional units of the processor and \( T_B \) be the number of cycles with a processor which can compute the computational primitive in a single cycle. Then, the maximum possible speed-up and efficiency can be defined as \( S_E = T_I/T_B \) and \( E_P = S_E/P \), respectively.

Table 2 gives the comparison of the various implementations of the 2-D FIR filter. In this table, the implementation I is based on the common computational primitive for the 2-D FIR filter, the implementation II is based on the common computational primitive for the 2-D IIR filter based on the scalar state space implementation, and the implementation III is based on the common computational primitive for the 2-D IIR filter based on the block state space implementation. For the implementation I, \( P = 3 \) as you can see in the figure 2. For the implementation II and III, \( P = 5 \) as shown in figure 3. The implementation III provides the highest speed-up and efficiency among them.

Table 3 gives the comparison of the implementation based on the computational primitive for the 2-D FIR filter (Implementation I) and the implementation based on the computational primitive for the 2-D IIR filter (Implementation II). The implementation II provides higher speed-up and efficiency.

Therefore, the implementation based on the computational primitive for the 2-D IIR filter with the provision for the block state space implementation gives the best result in terms of speed-up and efficiency for both 2-D IIR and FIR filters. Because of the high efficiency and speed-up, we can implement a real-time 2-D IIR and FIR filter with smaller number of processors than other implementations.

**Table 2: Performance comparison for the implementation of 2-D FIR filter**

<table>
<thead>
<tr>
<th>Order</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_I )</td>
<td>17</td>
<td>49</td>
<td>161</td>
<td>577</td>
<td>2177</td>
</tr>
<tr>
<td>( I-I_P )</td>
<td>9</td>
<td>25</td>
<td>81</td>
<td>289</td>
<td>1089</td>
</tr>
<tr>
<td>( I-S_P )</td>
<td>1.89</td>
<td>1.06</td>
<td>1.99</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>( I-E_P )</td>
<td>0.63</td>
<td>0.65</td>
<td>0.66</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>( II-T_B )</td>
<td>9</td>
<td>25</td>
<td>81</td>
<td>289</td>
<td>1089</td>
</tr>
<tr>
<td>( II-S_P )</td>
<td>1.89</td>
<td>1.06</td>
<td>1.99</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>( II-E_P )</td>
<td>0.38</td>
<td>0.39</td>
<td>0.40</td>
<td>0.40</td>
<td>0.40</td>
</tr>
<tr>
<td>( III-T_B )</td>
<td>6</td>
<td>15</td>
<td>45</td>
<td>153</td>
<td>561</td>
</tr>
<tr>
<td>( III-S_P )</td>
<td>2.83</td>
<td>3.27</td>
<td>3.58</td>
<td>3.77</td>
<td>3.88</td>
</tr>
<tr>
<td>( III-E_P )</td>
<td>0.57</td>
<td>0.65</td>
<td>0.72</td>
<td>0.75</td>
<td>0.78</td>
</tr>
</tbody>
</table>

**Table 3: Performance comparison for the implementation of 2-D IIR filter**

<table>
<thead>
<tr>
<th>Order</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_I )</td>
<td>33</td>
<td>97</td>
<td>321</td>
<td>1153</td>
<td>4353</td>
</tr>
<tr>
<td>( I-I_P )</td>
<td>15</td>
<td>50</td>
<td>162</td>
<td>578</td>
<td>2178</td>
</tr>
<tr>
<td>( I-S_P )</td>
<td>1.89</td>
<td>1.94</td>
<td>1.98</td>
<td>1.99</td>
<td>2.00</td>
</tr>
<tr>
<td>( I-E_P )</td>
<td>0.61</td>
<td>0.65</td>
<td>0.66</td>
<td>0.66</td>
<td>0.67</td>
</tr>
<tr>
<td>( II-T_B )</td>
<td>9</td>
<td>25</td>
<td>81</td>
<td>289</td>
<td>1089</td>
</tr>
<tr>
<td>( II-S_P )</td>
<td>3.67</td>
<td>3.68</td>
<td>3.96</td>
<td>3.99</td>
<td>4.00</td>
</tr>
<tr>
<td>( II-E_P )</td>
<td>0.73</td>
<td>0.78</td>
<td>0.79</td>
<td>0.80</td>
<td>0.80</td>
</tr>
</tbody>
</table>

**CONCLUSION**

In this paper, we presented an efficient hardware implementation for the spatial domain IIR and FIR digital filters based upon the state space representation. We presented a multiprocessor system which uses both algorithm partitioning and data partitioning to achieve real-time operation. With this system, we can achieve almost linear speed-up as we add additional processors until we reach real-time operation.

We derived two common computational primitives for the efficient implementations of the both filters. The processor can be implemented based upon either of those two computational primitives. We suggested the use the block state space implementation to derive one of the computational primitives. This is used for the efficient implementation of the 2-D FIR filter with the processor which is designed based upon the computational primitive for the 2-D IIR filter. An attractive of the approach presented in this paper is that it can be easily extended to the higher order systems or the higher dimensional algorithms.

**REFERENCES**


