

A new VLSI system for adaptive recursive filtering *

Kam Hoi CHENG ** and Sartaj SAHNI

Computer Science Department, 136 Lind Hall, University of Minnesota, Minneapolis, MN 55455, U.S.A.

Received July 1986

Abstract. We develop an efficient bidirectional chain VLSI system for the adaptive recursive filtering problem. Our design is an improvement over previous designs. It matches the performance of a broadcast chain but does not use the broadcast capability.

Keywords. VLSI architectures, systolic systems, adaptive recursive filtering.

1. Introduction

VLSI architectures for a variety of problems have been proposed by several authors. A bibliography of over 150 research papers dealing with this subject appears in [6]. In this paper, we are concerned solely with the adaptive recursive filtering problem. The input to this problem is an $n \times w$ matrix A of weighting coefficients and a $1 \times w$ vector (x_{1-w}, \dots, x_0) . The output is a $1 \times n$ vector (x_1, \dots, x_n) where

$$x_i = \sum_{j=1}^w a_{ij} x_{i+j-w-1}, \quad i = 1, 2, \dots, n. \quad (1)$$

In evaluating a VLSI design, we assume that the VLSI system will be attached to the host processor using a bus as in Fig. 1. The evaluation of a VLSI design should take the following into account:

- (1) *Processors*: how many processors are used in the VLSI system? This figure is denoted by P .
- (2) *Bus bandwidth*: the maximum amount of data to be transmitted between the host and the VLSI system in any cycle. This figure is denoted by B .
- (3) *Speed*: how much time does the VLSI system need to complete its task? This time may be decomposed into the times T_C (time for computations) and T_D (time for data transmissions both within the VLSI system and between the host and the VLSI system).

Let C denote the time spent for computation by a single processor algorithm and D denote the total amount of data that needs to be transmitted between the host and VLSI system. As an example, consider the problem of multiplying two $n \times n$ matrices A and B to get Y . Each element of Y is the sum of n products. We shall count one multiplication and addition as one arithmetic (or computation) step. If the classical matrix multiplication algorithm is used, $C = n^3$. If $P = n$, then $T_C \geq n^2$. The host needs to send $2n^2$ elements to the VLSI system and

* This research was supported in part by the National Science Foundation under grant MCS-83-05567.

** Currently with the Computer Science Department, University of Houston, Houston, TX, U.S.A.

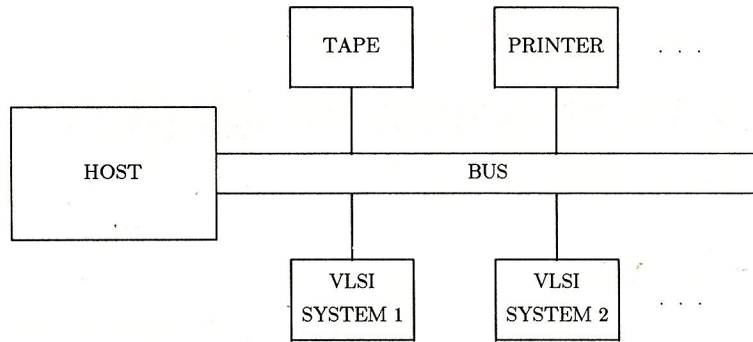


Fig. 1.

receive n^2 elements back. So, $D = 3n^2$. With a bandwidth of n , T_D must be at least $3n$. T_D will exceed $3n$ if the bandwidth is not used to capacity at all times. For the adaptive recursive filtering problem, $C = nw$ and $D = nw + n + w$. If n processors are used, then $T_C \geq w$ and $T_D > w + 1$.

The ratio

$$R_D = B * T_D / D$$

measures the effectiveness with which the bandwidth B has been used. Clearly, $R_D \geq 1$ for every VLSI design.

The ratio

$$R_C = P * T_C / C$$

measures the effectiveness of processor utilization. Once again, we see that $R_C \geq 1$ for every VLSI design.

Finally, we may combine the two efficiency ratios R_C and R_D into the single ratio $R = R_C * R_D$. A design that makes effective use of the available bandwidth and processors will have R close to 1.

The efficiency measure R as defined here is the same as that used in [1-3] to evaluate VLSI designs for matrix multiplication and back substitution. This measure is also quite similar to that proposed in [4]. In fact, the two measures become identical when $T_C = T_D$.

In comparing different architectures for the same problem, one must be wary about over emphasizing the importance of R_C , R_D and R . Clearly, by using $P = 1$ and $B = 1$, we get $R_C = R_D = R = 1$ but no speed up at all. So, we are really interested in minimizing T_C and T_D while keeping R close to 1.

VLSI architectures for the adaptive recursive filtering problem have been proposed earlier in [4,5,7-9]. The design of [4] uses a broadcast chain and has $P = w$, $B = w + 2$, $T_C = n + w - 1$, $T_D = n + w$, $R_C \sim 1 + w/n \sim 1$, $R_D \sim 1 + 1/w + w/n \sim 1$ and $R \sim 1$. The design of [5] uses a bidirectional chain of processors. An improved version is described in [8]. For this, $P = \lceil w/2 \rceil$, $B = \lceil w/2 \rceil + 2$, $T_C = 2n + w - 2$, $T_D = 2(n + w - 1)$, $R_C \sim 1 + w/(2n) \sim 1$, $R_D \sim 1 + 3/w + w/n \sim 1$ and $R \sim 1$. The design of [7] uses a systolic ring architecture to solve the simple recurrence problem. It can be easily extended to solve the adaptive recursive filtering problem. This extension has $P = \lceil w/2 \rceil$, $B = \lceil w/2 \rceil + 1$, $T_C = 2(n - 1) + w$, $T_D = 2(n + w - 1) + 1$, $R_C \sim 1 + w/(2n) \sim 1$, $R_D \sim 1 + 1/w + w/n \sim 1$ and $R \sim 1$.

While all the above designs have an $R \sim 1$, the broadcast chain of [4] has a T_C and T_D that is about half that of the other designs. In this paper, we develop a bidirectional chain VLSI system that has the same (actually slightly smaller) T_D and T_C as the broadcast chain of [4]. For our design, $P = w$, $B = w + 1$, $T_C = n + \lceil w/2 \rceil$, $T_D = n + w + 1$, $R_C \sim 1 + w/(2n) \sim 1$,

$R_D \sim 1 + w/n \sim 1$ and $R \sim 1$. Our design shows that a broadcast chain is not required to obtain this T_C and T_D performance.

2. $o(n)$ throughput bidirectional chain

An $o(n)$ throughput bidirectional chain for the adaptive recursive filtering problem can be obtained by extending the systolic design of [9] for the nonadaptive recursive filtering problem. This extension requires us to recast (1) into the following form:

$$\begin{aligned}
 x_i &= \sum_{j=1}^w a_{ij} x_{i+j-w-1} \\
 &= \sum_{j=1}^{w-1} a_{ij} x_{i+j-w-1} + a_{iw} \sum_{j=1}^w a_{i-1, j} x_{i+j-w-2} \\
 &= \sum_{j=1}^{w-1} a_{ij} x_{i+j-w-1} + a_{iw} \sum_{j=0}^{w-1} a_{i-1, j+1} x_{i+j-w-1} \\
 &= \sum_{j=0}^{w-1} b_{ij} x_{i+j-w-1}, \quad i \geq 1
 \end{aligned} \tag{2}$$

where

$$b_{i0} = a_{iw} a_{i-1, 1}, \quad b_{ij} = a_{ij} + a_{iw} a_{i-1, j+1}, \quad 1 \leq j \leq w-1, \tag{3}$$

$$a_{01} = 1, \quad a_{0j} = 0, \quad 2 \leq j \leq w, \quad x_{-w} = x_0. \tag{4}$$

To calculate the b_{ij} 's of (3) dynamically, w PEs in addition to the $w+1$ PEs used in [9] are needed. The performance figures of the resulting VLSI system are $P = 2w + 1$, $B = 2w + 3$, $T_C \sim n + \lceil w/2 \rceil$, $T_D \sim n + w$, $R_C \sim 2 + 1/w + w/n \sim 2$, $R_D \sim 2 + 1/w + w/n \sim 2$ and $R \sim 4$.

Improved performance can be obtained by using the bidirectional chain architecture of Fig. 2. Each PE has the ability to add, multiply, and transfer data to/from its left and right adjacent processor. All the even numbered PEs are on the left, while all the odd numbered PEs are on the right. The output is generated from the middle PE, PE(w). The PEs to the left of PE(w) compute all terms involving even columns of A , while PEs on the right compute all terms involving odd columns of A . The case when w is odd is shown in Fig. 2(a). The case when w is even is shown in Fig. 2(b).

The middle processor, PE(w), has the five registers: A , V , X , Y and Z . The remaining PEs have three registers (A , X and Y) each. We use the notation $R(i)$ to denote register R , $R \in \{A, V, X, Y, Z\}$, of PE(i). The A register of each PE is used to hold an input value from the A matrix. PE(i) receives input from column i of A only, $1 \leq i \leq w$. The X register of each PE holds an x_i value while the Y registers hold partial sums in the computation of an x_i . In each cycle, the $X(i)$'s move one step away from the center PE, PE(w), while the $Y(i)$'s move one step towards this PE.

The working of the VLSI system is described formally in Algorithm 2.1. The first **for** loop sets up the initial configuration. The three steps in the **parallel do** are executed simultaneously. When this **for** loop terminates, PE(w) contains x_p for $p = \lceil (w-1)/2 \rceil - w = \lfloor -(w+1)/2 \rfloor$ in its X register. The X register of a PE that is a units away from PE(w) contains x_{p-a} . The second **for** loop contains two sets of concurrently executed statements. In the first set, i.e. first **parallel do**, essentially five concurrent activities are performed in each iteration of this loop:

- (1) PE(w) either inputs an x_i , $i \leq 0$ or outputs a newly computed x_i , $i > 0$.
- (2) All X values move one PE away from the middle PE.

