A PIPELINE ARCHITECTURE FOR COMPUTING CUMULATIVE HYPERGEOMETRIC DISTRIBUTIONS*

Xiaobo Li

Department of Computer Science
Wichita State University
Wichita, KS 67208

Lionel M. Ni

Department of Computer Science
Michigan State University
East Lansing, MI 48824

Abstract: The hypergeometric distribution is a widely used arithmetic function and is fundamental to many statistical sampling and statistical pattern recognition problems. Computation of the cumulative hypergeometric distribution function, H(a), is extremely time-consuming. As a result, many approximation algorithms have been proposed to evaluate the cumulative hypergeometric distribution. This paper describes a two-level pipeline architecture for computing H(a) with computation complexity reduced to c+a, where c is a constant. The main part of the design is a type of recurrence computation. A modular and systematic approach is suggested to implement the recurrence formula. The computation complexity of the proposed architecture is also compared with various other known methods. The highly regular structure of the design can lead to efficient VLSI implementation.

I. INTRODUCTION

A finite population with N elements is classified into two groups. The probability of drawing the n-th element from one of the two groups depends on the previously drawn n-1 elements. The <u>cumulative hypergeometric probability</u> is defined as the probability of obtaining at most a given number of elements of a certain group in a sample [4]. The finite population thus can be expressed as an N-bit binary vector with each element being either a 0 or a 1. Let n be the number of 1's in the vector, r be the number of draws, and i be the number of 1's drawn. The probability density function of the hypergeometric distribution, h(i), is defined as

$$h(i) = h(N,n,r,i) = \frac{\binom{n}{i}\binom{N-n}{r-i}}{\binom{N}{r}} \qquad (1)$$

where $0 \le i \le \min\{n,r\}$. The cumulative hypergeometric distribution function, H(a), is defined as

$$H(a) = \sum_{i=0}^{a} h(i) \qquad 0 \le a \le \min\{n,r\} \qquad (2)$$

The cumulative hypergeometric distribution function is a widely-used arithmetic function and is fundamental to many statistical sampling and statistical pattern recognition problems, such as cluster validity analysis [1], tree classifier design [13], image template matching [10], feature analysis [9], and permutation statistics [5].

Direct computation of H(a) is extremely time consuming because the computation involves many long sequences of multiplication operations. Due to the limited computer word-length, the arithmetic evaluation sequence must be appropriately ordered to prevent the problem of arithmetic overflow from happening. To avoid this time-consuming computation, many normal and binomial approximation algorithms have been proposed to approximate H(a). A recent intensive comparison study by [11] has shown that the Peizer approximation is both accurate and simple. However, the relative error of the Peizer approximation could be up to 19.7% [11].

All of these previously mentioned algorithms are sequential and only provide approximate solutions. The advent of VLSI technology has triggered the idea of implementing arithmetic functions directly in hardware [8]. However, for direct VLSI implementation, these previously mentioned algorithms all seem to be too irregular or too complex to be useful. This paper describes a two-level pipeline architecture for computing H(a). The architecture discussed here is based on the concept of two-level pipelining and vector reduction [12]. In the next section, the computation complexities in evaluating H(a) using a direct implementation, a recursive method, and the Peizer approximation are discussed in that order, respectively. The proposed hardware architecture is based on a recursive method in evaluating H(a). Section III describes a modular and systematic approach in building the hardware architecture. The comparsion and evaluation of various architectural implementations are studied in Section IV. Section V provides some concluding remarks.

-----

* This research was supported by the NSF under grant ECS 83-04967.

CH2146-9/85/0000/0166$01.00 © 1985 IEEE

II. EVALUATION OF CUMULATIVE HYPERGEOMETRIC DISTRIBUTIONS

This section is concerned with various ways of computing H(a). The computation time, in terms of the number of clock cycles, is the sole performance index for comparing the different evaluation methods. We first define the following notations.

- t_a: number of cycles needed in performing addition.
- t_m: number of cycles needed in performing multiplication.
- t_d: number of cycles needed in performing division.
- t_r: number of cycles needed in performing the square root operation.
- t_g: number of cycles needed in performing the logarithm operation.
- t_t: number of cycles needed in looking up a standard normal distribution table.

Note that all arithmetic operations are floating-point operations.

A. Direct Evaluation

In computing H(a), the order of performing arithmetic functions is important because the word length is limited. In directly evaluating Eq.(1), the ratio of two long products should be done by alternating the division and multiplication operations. Since the direct computation involves factorials and a great deal of repetitive operations, it is very time consuming. In general, to compute C(a,b) (C is the binomial coefficient), it needs min{b,a-b} divisions and multiplications, respectively, for $0 \leq b \leq a$. Thus, the number of cycles required in computing H(a), T(direct), is data dependent. However, the lower bound has been found to be [10]

$$T(direct) \geq a(r+a)(t_m+t_d). \qquad (3)$$

It is clear that the direct evaluation of H(a) has a lower bound computational complexity of O(a(a+r)).

B. Recursive Evaluation

Direct evaluation of h(i) involves too many arithmetic operations and some of the results are repetitively computed. Another approach to evaluate h(i) is to use a recusive formula as defined below.

$$h(0) = \frac{m(m-1)\dots(m-r+1)}{N(N-1)\dots(N-r+1)} \quad \text{where } m=N-n \qquad (4)$$

and

$$h(i) = h(i-1)X(i-1) \quad \text{where}$$

$$X(i-1) = \frac{(n-i-1)(r-i-1)}{i(N-n-r+i)} \qquad (5)$$

It takes r divisions and (r-1) multiplications to evaluate h(0). For the remaining h(i), $0<i\leq a$, it needs two multiplications and two divisions if the division operation is performed first in Eq.(5). Thus, the total number of clock cycles needed in evaluating H(a) using the recursive formula will be

$$T(recur) = (r-1)t_m + rt_d + a(2t_m+2t_d+t_a) \qquad (6)$$

The computation complexity of the recursive evaluation is O(a+r), which is reduced from the direct evaluation.

C. The Peizer Approximation

Even the recursive evaluation of H(a) is time-consuming when N, r, a, and n are large numbers, which are usually the case. In programming the evaluation of Eqs.(4) and (5), in addition to these basic arithmetic operations, there are considerable amount of time spent in indexing, looping, and performing other software overhead. Thus, many approximation methods for computing H(a) have been investigated by many researchers. A recent extensive empirical study of the accuracy of twelve normal and three binomial approximations has shown that a normal approximation by Peizer is both far superior to other approximations and is simple to compute [11].

First, let us briefly state the Peizer formula and give a simple discussion on its computational complexity. When using the Peizer approximation, H(a) is approximated by a standard normal distribution with the cumulative distribution function expressed as $F_n(z)$, i.e, $H(a) \approx F_n(z)$. The parameter z is defined below as:

$$z = \frac{A''D'' - B''C''}{|AD - BC|} \sqrt{\frac{2mnrsN'L}{m'n'r's'N}}$$

where A=a+0.5, A'=A+(1/6), A"=A'+[0.02/(A+0.5)]+[0.01/(n+1)]+[0.01/(r+1)], and B, B", C, C", D, and D" are defined in a similar manner with b=n-a, c=r-a, and d=N-n-c, respectively. Also, m=N-n, s=N-r, N'=N-(1/6), and L is defined as L = A[log(AN/nr)] + B[log(BN/ns)] + C[log(CN/mr)] + D[log(DN/ms)]. The error rate of this approximation varies with respect to a. For $a \geq 2$, the maximum absolute error of this approximation is .001 and the maximum relative (percentage) error ranges from 0.71% to 19.7%. The guaranteed number of correct decimal places in this case is at least 3.040 [11]. The computation time of Peizer approximation, T(Peizer), is a constant and is independent of the values of N, r, a, and n. The number of arithmetic operations needed will be [10]

$$T(Peizer)=38t_a+22t_m+18t_d+4t_g+t_r+t_t \qquad (7)$$

Note that the evaluation of the Peizer approximation includes the time needed to access normal distribution table after z is derived.

III. HARDWARE IMPLEMENTATION

The advent of VLSI technology has triggered the idea of directly implementing arithmetic functions in hardware. Several factors must be considered in designing a hardware architecture. First, the I/O bottleneck must be avoided. Second, the hardware functional unit must be efficiently utilized. Third, the interconnection pattern among the functional units must be regular or simple. Fourth, the extra buffer space for intermediate storage must be eliminated. The direct evaluation method involves much more arithmetic operations than the recursive method. Furthermore, hardware implementation of the direct evaluation method needs either extra buffers for intermediate storage or excessive functional units. Thus, we tried to implement the recursive method in hardware.

The overall structure of the design is shown in Figure 1. There are four computing blocks: X(i) generation, Y(i) generation, h(i) generation, and H(a) generation. The design is based on the concept of two-level pipelining which is pipelining between blocks and pipelining within each block. The input parameters are N, n, r, a, and h(0). The initial value of h(0) can be computed either by a straightforward hardware device or by a table look-up. The MUX (multiplexer) boxes are needed to select an appropriate input to the pipeline units. In the following subsections, we will describe the function of each of the four computing blocks diagramed in Figure 1.

A. Compute H(a)

The H(a) generation block contains a pipeline adder with $z=t_a$ segments and with a feedback path as shown in Fig. 1. After a certain period of time, the h(i) generation block will generate h(0), h(1), h(2), ..., up until h(a) in sequence. Each will be produced one per cycle. The pipeline adder tries to accumulate these h(i)'s. The H(a) generation block is a typical pipelined vector reduction unit [12]. During the first z cycles after h(0) is generated, the multiplexer will select "0" input. After z cycles, the output of the pipeline adder will be selected by the multiplexer. After h(a) is generated, these a+1 scalars of h(i)'s will be partitioned into z groups where each group has a partial sum of some h(i)'s. To merge these z partial sums into a final sum, which is H(a), either a symmetric merging method or an asymmetric merging method may be used [12]. For the asymmetric method, the time needed to merge these partial sums will be

$$t_{merge} = \begin{cases} (z-1)+z\lceil \log_2 z\rceil - 2^{\lceil \log_2 z\rceil} + z & \text{if } (a+1)\geq z \\ (z-1)+(a+1)\lceil \log_2 (a+1)\rceil - 2^{\lceil \log_2 (a+1)\rceil} & \\ +(a+1)+(z-a-1)\lceil \log_2 (a+1)\rceil & \text{if } (a+1)<z \end{cases} \quad (8)$$

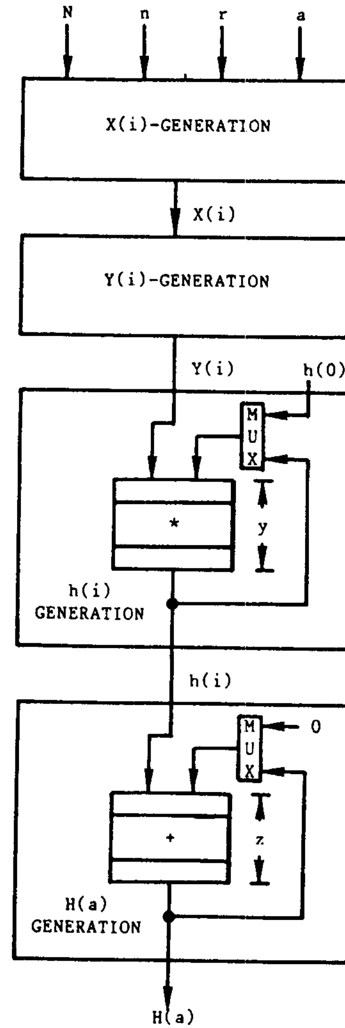

Fig. 1. An overall architecture for computing H(a)

B. Compute h(i)'s

The h(i) generation block contains a pipeline multiplier with $y=t_m$ segments. Since h(i) is defined recursively as shown in Eq.(5), a feedback path is needed as indicated in Fig. 1. According to Eq.(5), h(i) depends on the value of h(i-1). Direct implementation of this dependence relationship implies that only one segment among y segments will be utilized. To fully utilize the pipeline multiplier, Eq.(5) can be rewritten as a companion function

$$h(i) = h(i-1)X(i-1)$$

$$= \begin{cases} h(0)X(i-1)X(i-2)\ldots X(0) & \text{for } 1\leq i\leq y \\ h(i-y)X(i-1)X(i-2)\ldots X(i-y) & \text{for } y\leq i\leq a \end{cases}$$

$$= \begin{cases} h(0)Y(i) & \text{for } 1 \le i \le y \\ h(i-y)Y(i) & \text{for } y \le i \le a \end{cases} \quad (9)$$

where

$$Y(i) = \begin{cases} X(i-1)*X(i-2)*\ldots*X(0) & \text{for } 1 \le i \le y \\ X(i-1)*X(i-2)*\ldots*X(i-y) & \text{for } y \le i \le a \end{cases} \quad (10)$$

Thus, when Y(i) is generated by the Y(i) generation block, the output of the pipeline multiplier, which is h(i-y), will be fed back to the input of the pipeline multiplier and operate with Y(i) as indicated in Eq.(9). Note that for Y(1) to Y(y), the MUX will select h(0) as the input instead of the output of the multiplier. The time delay for generating h(i) after Y(i) is computed will be y cycles.

C. Compute X(i)'s

The X(i) generation block, based on Eq.(5), produces a sequence of X(i)'s for i ranging from 0 to a. Figure 2 shows the hardware architecture in computing X(i). The small boxes with a "decrement" or "increment" control input are counters. The initial value of each counter is indicated by the input labelled at the top of each box. Two pipeline multipliers and one pipeline divider are used to implement Eq.(5). After t_m+t_d cycles, the first output X(0) will be generated. Then the successive X(i)'s will be continuously generated in every cycle. The AND gate with clock and enable inputs will make sure that X(0) up to X(a) will be generated.

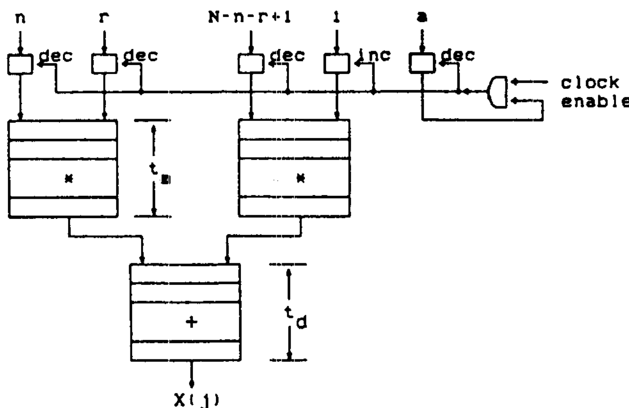

Fig. 2. Hardware architecture for computing X(j)

D. Compute Y(i)'s

Y(i) is defined in Eq.(10). The value of y depends on the number of segments of the pipeline multiplier in the h(i) generation box. Equation (10) is actually a recurrence computation. In fact, the design can be applied to any operator "*" as long as the operator is associative and commutative.

In evaluating various values of Y(i), X(i)'s will be repetitively used. Again, Y(i)'s must be continuously generated, one per cycle, in order to fully utilized the pipeline functional units and to speedup the computation process. For this purpose, some dummy segments (or noncompute delays) are needed. A pipeline dummy unit (PDU) with one input and one output consists of a number of dummy segments which will introduce necessary delays for the purpose of synchronization. A pipeline functional unit (PFU) with two inputs and one output performs the arithmetic operation "*". In general, let s be the number of segments required in the PFU for evaluating "*". For the case of $y=2^m$ for some integer m, Y(i)'s can be computed by cascading m PFUs where one input of each PFU is inserted with a PDU. This design was originally proposed by Kogge [7]. Figure 3 shows an example for the case of y=4. Note that the number of segments of each pipeline unit is indicated within the box.

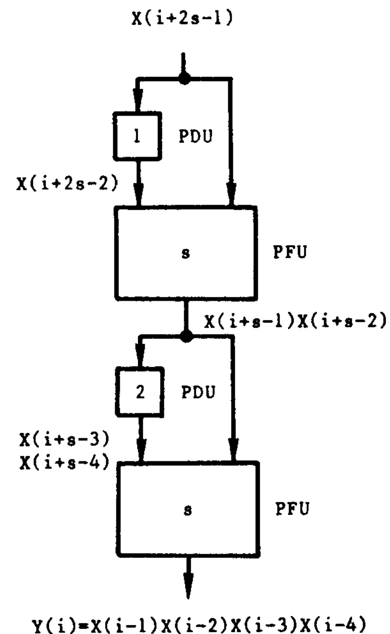

Fig. 3. Compute Y(i) from X(j) when y=4

Since the value of y may vary, we shall propose a systematic approach in the design of the Y(i) generation block for a given value of y. A modular and systematic approach is suggested by using two types of building cells. Figure 4 shows the internal design of these two building cells, Main-cell (M-cell) and Auxiliary-cell (A-cell), respectively. The control input k to the M-cell will determine the number of dummy segments, which is 2^k, introduced in the PDU. The control input y_k is to select one of the two inputs of the MUX. "u" is an unit element of the operator "*". If "*" is multiplication, then u is a constant 1.

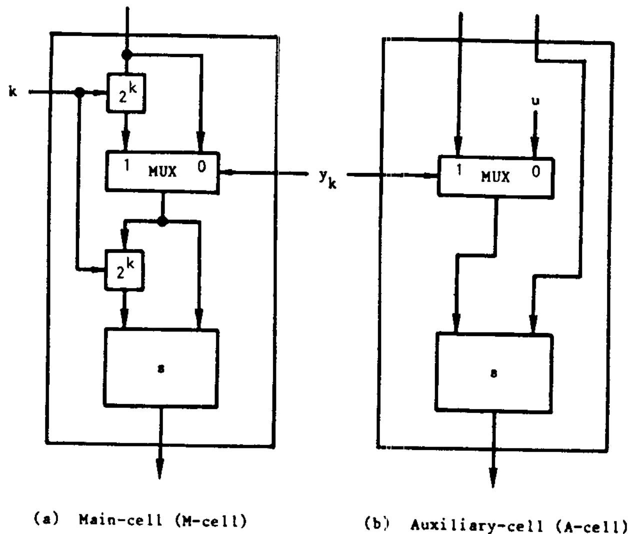

Fig. 4. The architecture of M-cell and A-cell

Given an arbitrary value of y, the Y(i)-generation block can be systematically constructed by using these cells. Let $y=y_{m-1}\cdots y_0$, where $y_{m-1}=1$. If $y=2^{m-1}$, then the Y(i)-generation block can be constructed by cascading (m-1) M-cells. In this case, $y_k=0$ for $0\leq k\leq m-2$. Thus, the first 2^k dummy segments are all skipped in each of the M-cells. As a consequence, the resulting architecture is similar to that of Fig. 3.

If $y<>2^{m-1}$, then it needs m cascading stages (k=0,...,m-1). In the first (m-1) stages (k=0,...,m-2), each stage has one M-cell and one A-cell connected in parallel. The last stage (k=m-1) has only one A-cell. Figure 5 demonstrates a design example for the case of y=13. Figure 5 also shows how a Y(i) is computed from those 13 X(j)'s, for j=i-1 down to i-13. Note that each cell will introduce at least s cycles of delay due to these s-segment PFUs. The extra delays introduced by the PDUs will perform the necessary synchronization functions. In the A-cell, if $y_k=1$, then both external inputs will go through the PFUs; otherwise, the left external input will not be used and an internally generated unit element "u" will be fed into the PFU. For demonstration purposes, the relative indices of the two inputs to the PFU are indicated by parentheses pairs inside each cell as shown in Fig. 5, where the one on the top is the left input to the PFU and the one on the bottom is the right input. The relative indices of the output of the PFU is indicated in the output port of the cell (or the input port of the next cell). Note that the output will be generated s cycles later.

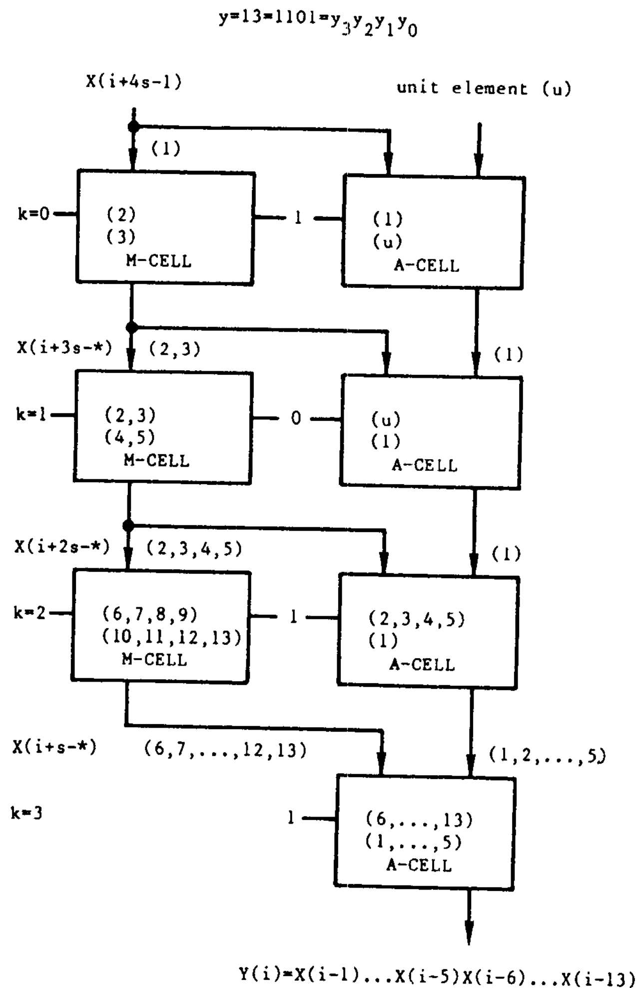

Fig. 5. An illustrative example for the case of y=13 using two types of building cells

Figure 5 shows the inputs and outputs of the PFUs of each cell at the instant when Y(i) is generated. Note that X(i+3s-*) at the stage k=1 means that all integer indices are relative to X(i+3s-*). For example, (2,3) in the M-cell of stage k=1 represents X(i+3s-2)X(i+3s-3). These A-cells contribute X(i-1) to X(i-5) as shown in Fig. 5. The three M-cells will contribute 8 X's from X(i-6) to X(i-13). In each M-cell, if $y_k=0$ then the first 2^k dummy segments will be skipped; otherwise, an extra delay of 2^k cycles will be introduced. For example, in stage k=2 with $y_2=1$, 4 extra cycles are introduced. Thus, the input (2,3,4,5) becomes (6,7,8,9) and is one input of the PFU. The other input (10,11,12,13) comes from (6,7,8,9) with 4 extra delay cycles introduced by the second PDU.

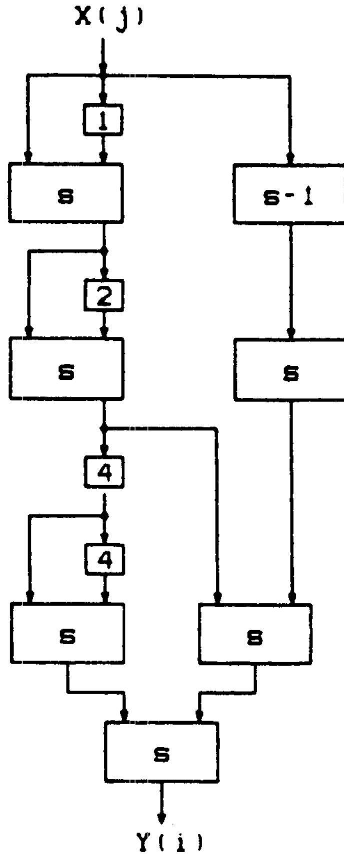

Fig. 6. An illustrative example for the case of y=13 by eliminating those redundant components from Fig. 5.

The modular design provides a systematic approach in constructing the Y(i)-generation block by using two types of cells. In general, for a given y, it needs $\log_2 y$ M-cells if y is an integer power of 2 and $\lfloor \log_2 y \rfloor$ M-cells and $\lceil \log_2 y \rceil$ A-cells, otherwise. Thus, the time interval, t_y, between feeding X(0) and getting Y(1) to/from the Y(i)-generation block will be

$$t_Y = s\lceil \log_2 y \rceil = t_m \lceil \log_2 t_m \rceil \qquad (11)$$

Note that if the Y(i)-generation block is designed individually for each y and without using these building cells, the design may be slightly simplified by eliminating those unused PDUs. Figure 6 shows an example for the case of y=13 again. The A-cell used in stage k=0 of Fig. 5 can be replaced by a (s-1)-segment PDU because its right input is always a unit element and the first 1-segment PDU in the left M-cell can be eliminated. The first PDU in the M-cell of stage-1 can also be eliminated because y_1=0. The time needed in filling up the Y(i)-generation block thus can also be slightly reduced.

IV. PERFORMANCE EVALUATION AND COMPARISON

We now derive the total computing time in evaluating H(a), which is denoted by T(pipe). It takes t_m+t_d cycles to generate X(0) from the X(i)-generation block. Y(1) will be generated $t_m\lceil \log_2 t_m \rceil$ cycles later from the Y(i)-generation block as indicated in Eq.(11). h(0) will be generated another t_m cycles later via the h(i)-generation block. It takes "a" more cycles to generate h(1) to h(a) in sequence. Thus, we have

$$T(pipe) = t_m + t_d + t_m \lceil \log_2 t_m \rceil + t_m + a + t_{merge} \qquad (12)$$

where t_{merge} is defined in Eq.(8) by replacing z to t_a. The computation complexity of the proposed pipeline architecture is clearly O(a).

We compare the computation times of all methods, T(direct), T(recur), T(Peizer), and T(pipe), as indicated in Eqs. (3), (6), (7), and (12), respectively. Note that the computation time only involves the arithmetic evaluation time. Programming overhead needed in evaluating the other three methods, such as looping and indexing, is not considered.

To give a better feeling in comparing those different methods, a numerical example is demonstrated based on the architecture of CRAY-1 [2]. Thus, we have t_a=6, t_m=7 and t_d=29. The logarithmic operation is performed by table lookup, which takes t_g=t_t=4 cycles. The square root operation assumes a standard algorithm [6] with t_r=30 cycles. Table 1 shows the number of cycles required for each method based on these figures. Note that the computing time of the Peizer approximation is problem dependent. Therefore, this approximation is preferred when "a" is extremely large and the exact value of H(a) is not necessary. Of course, one may wish to implement Peizer's approximation in hardware and to exploit all possible parallelism. We have found that the hardware implementation of Peizer's approximation will need T(Hardware-Peizer) cycles [10].

$$T(Hardware\text{-}Peizer) = 3t_a + 4t_m + t_d + t_g + t_r + t_t \qquad (13)$$

However, there is a considerable amount of functional unit hardware involved in implementing Peizer's approximation and this will not gain too much in terms of the computation time compared with the pipeline implementation which provides an exact result.

V. CONCLUSIONS

In this study, we reviewed some existing techniques for computing the cumulative distribution function of the hypergeometric distribution. Since the direct computation is very time consuming, an

approximation is desired in many cases. The recursive algorithm for the exact computation avoids much of the repetition of the direct method. We proposed and presented a detailed hardware pipeline architecture which implements the recursive formula. The main part of the design is the Y(i)-generation block which is actually a general solution for a kind of recurrence computation. This pipeline architecture is computationally efficient, O(a), as compared to other exact computations, O(a(a+r)) and O(a+r). The modularity and the regularity of the system architecture make it suited for VLSI implementation.

Table 1. Comparison of computation times (cycles)

T(direct)	> 36a(a+r)
T(recur)	36r+78a-7
T(Peizer)	1032
T(pipe)	if $a \geq 5$ then a+85 if a<5 then $2a+70+6\lceil \log_2(a+1) \rceil - 2^{\lceil \log_2(a+1) \rceil}$

REFERENCES

[1] Bailey, T.A. Jr. and Dubes, R.C., "Cluster validity profiles," Pattern Recognition, Vol.15, pp.61-83, 1982.

[2] Cray research, Cray-1 Computer System Hardware Reference manual, 2240004, 1977.

[3] Fowlkes, E.G. and Mallows, C.L., "A method for comparing two hierarchical clusterings," J. of American Statistical Association, Vol.78, pp.553-569, September 1983.

[4] Hoel, P., Port, S. and Stone, C. Introduction to Probability Theory, Houghton Mifflin Pub. Co., Boston, 1971.

[5] Hubert, L., "Generalized proximity function comparisons," British J. of Math. Statist. and Psychol., Vol.31, pp.179-192, 1978.

[6] Hwang, K., Computer Arithmetic, John Wiley & Sons, New York, 1979.

[7] Kogge, P.M. The Architecture of Pipelined Computers, Chapter 2, McGraw-Hill Book Co., 1981.

[8] Kung, H.T., "Let's design algorithms for VLSI systems," Technical Report CMU-CS-79-151, Carnegie-Mellon University, Computer Science Department, January 1980.

[9] Li, X. and Dubes, R.C., "The selection of significant dichotomous features," Proc. of the 7th Int'l Conf. on Pattern Recognition, Montreal, Canada, pp.260-263, August 1984.

[10] Li, X., "A probabilistic association measure for pattern recognition," Ph.D. Dissertation, Department of Computer Science, Michigan State University, 155 pages, August 1984.

[11] Ling, R.F. and Pratt, J.W., "The accuracy of Peizer approximations to the hypergeometric distribution, with comparison to some other approximations," J. of American Statistical Association, Vol.79, No.385, pp.49-60, March 1984.

[12] Ni, L.M. and Hwang, K., "Vector-reduction techniques for arithmetic pipelines," accepted to appear in IEEE Trans. on Computers, May 1985.

[13] Tebbe, D.L., "A Multi-catehory decision network of dichotomous decision trees," Proc. of the 7th Int'l Conf. on Pattern Recognition, pp.264-266, 1984.