On Self-Routing in Beneš and Shuffle-Exchange Networks

C. S. Raghavendra and Rajendra V. Boppana

***Abstract*—In this paper, we present self-routing algorithms for realizing the class of linear permutations in various multistage networks such as Beneš, $2n$-stage shuffle-exchange, etc. Linear permutations are useful in providing fast access of data arrays. In the first half of the network, switches are set by comparing the destination tags at their inputs, and, in the second half, switches are set using the Omega self-routing algorithm. We show that the comparison operations can be implemented in bit-serial networks without loss of time. In contrast, with the well-known Beneš network self-routing algorithm of Nassimi and Sahni [10], switches are set by giving priority to the destination tag at the upper input to them. Their algorithm routes many useful permutations but the class of linear permutations. The previously known techniques to realize linear permutations in multistage networks are not of the self-routing type. Hence, the algorithms presented are extremely useful in providing fast access of various data patterns using interconnection networks cheaper than crossbars.**

***Index Terms*—Beneš network, interconnection networks, linear permutations, self-routing algorithms, shuffle-exchange networks.**

I. INTRODUCTION

Typically, a parallel processor consists of a number of processors and an interconnection network for exchange of information among them as well as with memory units. In a processor–memory network model, any processor should be able to communicate with any memory unit, which is called full access. To keep the communication step overhead minimum, parallel algorithms are often designed with permutation (one-to-one) type data transfers. To support SIMD type computations, ideally one would like the network to be able to perform all the permutations that allow simultaneous use of the memory units. If the underlying network cannot support a required permutation function, then it has to be realized in multiple passes through it. To avoid this, crossbar networks or networks that are rearrangeable, for example, the Beneš network, can be used as interconnection networks. The advantages with rearrangeable networks are any permutation can be realized in one pass through the networks, and, if they are built using smaller switches such as 2×2 switches, then they are cheaper than crossbar networks. Therefore, rearrangeable networks are used in some parallel computer implementations (e.g., GF-11 [1]).

A well-known rearrangeable network is the Beneš network [2], which is built in a recursive manner using 2×2 switches. (An 8×8 Beneš network constructed recursively from two 4×4 Beneš networks is shown in Fig. 1.) In such networks, it takes some time to set up the switches to realize a given arbitrary permutation. For an $N \times N, N = 2^n$, Beneš network $\mathcal{B}_n$, determining the switch settings, to realize an arbitrary permutation, takes $O(nN)$ time on a uniprocessor computer [17]. Parallel algorithms to determine the switch settings require $O(n^2)$ or $O(n^4)$ time using, respectively, a completely interconnected computer or a shuffle-exchange computer [11]. Therefore, if the required permutations change frequently during a computation, the communication time may become the bottleneck.

Manuscript received September 12, 1988; revised July 13, 1990. This work was supported by the NSF Presidential Young Investigator Award MIP 8452003, DARPA/ARO Contract DAAG 29-84-k-0066, ONR Contract N00014-86-k-06062.

The authors are with the Department of EE–Systems, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 9042300.

0018-9340/91/0900-1057$01.00 © 1991 IEEE

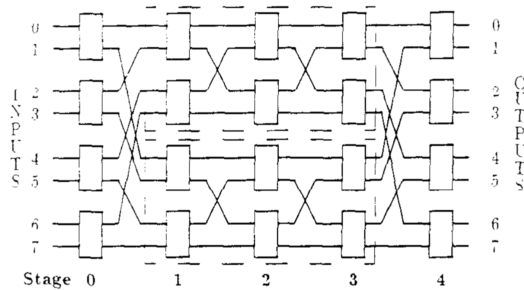

Fig. 1. An 8×8 Beneš network, $\mathcal{B}_3$. The dashed boxes indicate 4×4 Beneš networks.

Often, the above methods, which can realize arbitrary permutations in Beneš network, are unusable for the following reasons. The time to set up the network is large compared to the propagation delay of the network, which is $O(n)$. Since the status of each switch is decided elsewhere, for example, at a centralized network controller, and transmitted to the switches, excessive hardware costs and time delays are encountered with these methods. Also, these methods do not facilitate pipelining of data movement—desirable in reducing the overhead of communication by overlapping it with computations—in interconnection networks. Finally, due to the nature of techniques used in developing parallel algorithms, the permutations required are generally nice and regular and can be expressed as algebraic functions [8]. This motivates us to develop fast self-routing algorithms—which route permutations by setting up switches on-the-fly using only local information (source and destination information of the messages present) at each switch—for many useful permutations required in parallel processing, if not for all the $N!$ permutations.

Nassimi and Sahni [10] developed a self-routing algorithm to pass the class of the bit-permute-complement permutations in the Beneš network. Their algorithm also routes the Lenfant's FUB families [8], which are shown to occur in the execution of various parallel algorithms by Lenfant. Yew and Lawrie [18] adapted the Nassimi and Sahni's Beneš network routing algorithm to realize bit-permute-complement permutations in $2n$-stage shuffle-exchange network, $\prod_n$. Recently, Nassimi gave a generalized self-routing algorithm to realize bit-permute-complement permutations in a class of $(2n-1)$-stage networks [9]. However, none of these methods realizes the linear class of permutations, which are useful in the parallel access of data arrays [4].

In this paper, we develop self-routing algorithms for routing the linear class of permutations on various interconnection networks. These algorithms are simple and route many other classes of permutations as well. Our algorithms differ from those of Nassimi and Sahni [10] and Yew and Lawrie [18] in giving priority to route messages, when there is contention for the output links of a switch. Our algorithms give priority, based on some type of comparison operation, to the smaller of the two whereas their algorithms give priority to the message at upper input line of the switch. We consider Beneš, $\prod$, and $(2n-1)$-stage Shuffle-Exchange networks. Our results include simple routing algorithms for the classes of linear (we extend this class with complements of bits), Omega, and inverse Omega permutations on these networks. For other permutations one can use a general looping type algorithm or break the original permutation into multiple simpler permutations.

II. Preliminaries

Interconnection Networks: We consider $N \times N, N = 2^n$ with $n \geq 2$, Beneš, $\prod$, and $(2n-1)$-stage Shuffle-Exchange networks constructed of 2×2 crossbars (switches). These networks may be used for processor-to-processor or processor-to-memory interconnections. The $2n$-stage Shuffle-Exchange network, $\prod_n$, is a cascade of two copies of Lawrie's Omega network (Ω_n), the n-stage Shuffle-Exchange network. ($\prod_3$ is shown in Fig. 2.) The $(2n-1)$-stage shuffle-exchange network is theoretically interesting due to the long standing conjecture about its rearrangeability [16].

The input ports, also the output ports, of a network are numbered $0, \cdots, N-1$, top to bottom. The stages of a network of k stages are numbered, from left to right, $0, \cdots, k-1$. Switch (i, j) is the jth switch, numbered top to bottom $0, \cdots, \frac{N}{2}-1$, in ith stage. Addresses of lines within a network are assigned as follows. An input line to a switch in stage 0 has the same address as that of the network input port to which it is connected. If the upper and lower input lines of a switch are indexed a and b, respectively, then its upper and lower output lines are also indexed a and b, respectively. Interconnection patterns do not affect the addresses of lines. This numbering is illustrated in Fig. 2 for $\prod_3$.

The multistage networks we consider share a property. In a stage, addresses of inputs to each switch differ in exactly one common bit, called the *connecting* bit; the upper line address is smaller compared to the lower line address. For $\mathcal{B}_n$, connecting bits for stages $0, \cdots, n-2, n, \cdots, 2n-2$ are, respectively, $0, \cdots, n-2, n-1, n-2, \cdots, 0$. And, for a k stage shuffle exchange network, the connecting bits are $n-1, \cdots, n-k \pmod n$, for stages, respectively, $0 \cdots, k-1$.

Notation: Each input (respectively, output) line is given a unique and distinct index x, $0 \leq x < N$, which may be represented in binary form as $x_{n-1} \cdots x_0$ with x_{n-1} being the most significant bit (MSB). However, it is treated as an n-bit column vector $(x_0, \cdots, x_{n-1})^T$ (the superscript T indicates the matrix transpose operation) in the Boolean matrix-vector computations. Similarly, given an n-bit column vector $(y_0, \cdots, y_{n-1})^T$, its value is computed as $\sum_{i=0}^{n-1} y_i 2^i$. Given a Boolean matrix $Q = (q_{i,j})_{n \times n}$, the matrix-vector product Qx is the n-bit vector given below. Here "$\dotplus$" indicates modulo 2 addition of bits.

$$\begin{pmatrix} x_0 q_{0,0} \dotplus \cdots \dotplus x_{n-1} q_{0,n-1} \\ \vdots \\ x_0 q_{n-1,0} \dotplus \cdots \dotplus x_{n-1} q_{n-1,n-1} \end{pmatrix}.$$

If the columns of Q are represented as $q_0, \cdots, q_{n-1}$, then $Q = (q_0, \cdots, q_{n-1})$ and $Qx = x_0 q_0 \dotplus \cdots \dotplus x_{n-1} q_{n-1}$. Here, $x_i q_i$ indicates the multiplication of each component of q_i with the scalar x_i, and $x_i q_i \dotplus x_j q_j$ indicates componentwise module 2 addition of the vectors $x_i q_i$ and $x_j q_j$.

Linear permutations: Let $V = \{0, 1, \cdots, N-1\}$. A linear permutation on V is a permutation that maps each $x \in V$ to some $y \in V$ such that each bit in the binary form of y is a linear combination of the bits of x. (Addition of bits is modulo 2.)

Definition 1: A permutation on V is said to be linear [13], if there exists an $n \times n$ binary matrix Q such that, for every $x \in V$, its image y is given by the following equation

$$y = Qx. \tag{1}$$

Note that Q is nonsingular by definition. If each bit of y is a linear combination of the bits of x and constant 1, then the resulting permutation is a linear-complement permutation. Formally, this is defined as follows.

Definition 2: A permutation on V is a linear-complement permutation, if there exists an $n \times n$ binary matrix Q and an n-bit column vector c such that, for every $x \in V$, its image y is given by the

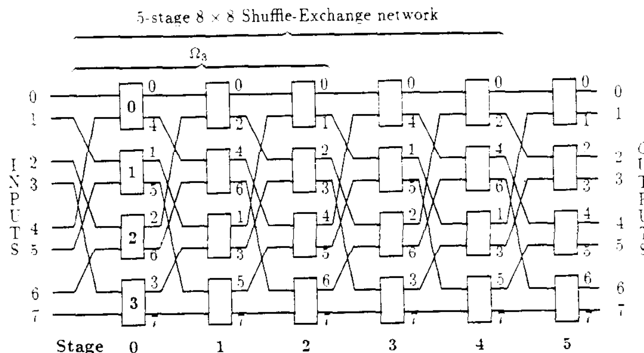

Fig. 2. A 6 stage 8 × 8 Shuffle-Exchange network. Numbers within a switch, shown only for stage 0, indicate its number in that stage. Numbers on the lines represent their addresses.

following equation.

$$y = Qx \oplus c. \tag{2}$$

In the literature [3], [6], linear permutations are termed as the nonsingular linear transformations of the n-dimensional vector space over the field $GF(2)$—the field consisting of two elements: 0 and 1.

The Boolean matrix Q in (2) is the important parameter of a linear-complement permutation. Any two linear-complement permutations with the same "Q" have similar properties. Given any $\pi_1, \pi_2 \in \mathcal{LC}_n$, $\pi_1^{-1} \in \mathcal{LC}_n$, and the composed permutation (right to left) $\pi_2\pi_1 \in \mathcal{LC}_n$.

If Q is a nonsingular lower (respectively, upper) triangular matrix, then the corresponding permutation is called a lower (respectively, upper) triangular linear-complement permutation or simply a lower (respectively, upper) triangular linear permutation, for $c = 0$. If a permutation matrix—a nonsingular matrix with exactly one 1 in each row and column—is chosen as Q in (2), then the resulting linear-complement permutation is also a bit-permute-complement permutation [10]. For any $N = 2^n$, $n \geq 2$, the class of bit-permute-complement permutations $(\mathcal{BPC}_n)$ is a special subclass of the class of linear-complement permutations $(\mathcal{LC}_n)$. There are $n!N$ and $2^{n(n+1)/2}\prod_{i=1}^{n}(2^i - 1)$ permutations in $\mathcal{BPC}_n$ and $\mathcal{LC}_n$, respectively; for $N = 8$, $|\mathcal{BPC}_3| = 48$ and $|\mathcal{LC}_3| = 1344$, where $|X|$ is the cardinality of set X.

Definition 3: The class of permutations passable by Omega network is called the Omega class. And the class of inverse Omega passable permutations is called the inverse Omega class.

Lawrie [7] and Pease [13] characterized these two classes of permutations; a permutation is in Omega or inverse Omega class if and only if each bit of a tag (y) is a function of the bits of its input line (x) given by (3) or (4), respectively. Both these classes contain lower and upper triangular linear-complement permutations.

$$y_i = x_i \oplus f_i(y_{n-1}, \cdots, y_{i+1}, x_{i-1}, \cdots, x_0), \qquad 0 \leq i < n \tag{3}$$

$$y_i = x_i \oplus f_i(y_{n-1}, \cdots, y_0, x_{n-1}, \cdots, x_{i+1}), \qquad 0 \leq i < n. \tag{4}$$

Here, f_i's are arbitrary Boolean functions.

III. The Self-Routing Algorithms

In permutation routing, each input of the network has the address of a unique and distinct output line, called the destination address or, simply, the tag. With a self-routing control algorithm, each switch in the interconnection network uses the tags at its inputs and sets itself appropriately using some simple logic.

The well-known bit-controlled (variously called, Omega, Delta, or digit-controlled) self-routing algorithm routes a tag as follows. If the ith bit is the connecting bit for a stage, then the ith bit of the tag, called *routing* bit, is used for routing it through that stage; it is routed to the upper (lower) output line of the switch, if the routing bit is 0 (1).

When the routing bits of both the tags at a switch are the same, there exists a conflict in setting up the switch since both of them specify the same output line. Some important permutations which exhibit switch conflicts are the bit reversal, shuffle, etc. Even though these permutations can be realized by Beneš and shuffle-exchange networks with at least $2n - 1$ stages, the simple bit-controlled algorithm cannot route them. Therefore, to route such permutations in Beneš or other networks, the above algorithm has to be modified suitably by giving priority to one of the input lines in setting up a switch whenever there is conflict.

Nassimi and Sahni [10] proposed to resolve such conflicts by giving priority to the upper input line; that is, whenever there is a conflict in setting up a switch, the tag at the upper input line is routed to the output line specified by its routing bit and the tag at the lower input line is routed to the remaining output line. Their method is simple, yet very powerful; it can realize the important classes of bit-permute-complement permutations, inverse Omega permutations, and many others. However, the class of linear-complement permutations cannot be realized by their method.

Pease [13] showed that a given linear permutation can be realized in two passes through the indirect binary n-cube network (identical to inverse Omega network [12]), by decomposing it into two simpler permutations, each of which is realized by the network in a single pass. However, this is not a self-routing method, since it involves the factorization of the Boolean matrix Q corresponding to the linear permutation. Etzion and Lempel [5] describe a method to pass linear permutations in $(2n - 1)$-stage Shuffle-Exchange networks, but it is not of self-routing type and takes $O(n^2)$ time to set up the network.

Given below are simple methods to pass linear-complement permutations in Beneš and Shuffle-Exchange networks. They are quite powerful and realize many other permutations as well.

A. An Algorithm for the Beneš Network

Algorithm BL: For the first $(n - 1)$ stages of $\mathcal{B}_n$, switches are set up such that input line with smaller destination tag value is routed according to its routing bit. For the next n stages, switches are set up using the standard Omega self-routing algorithm. ∎

This algorithm is different from that of Nassimi and Sahni [10]. In case of conflict in setting up a switch, their algorithm gives priority

Fig. 3. An example showing switch settings done by Algorithm BL.

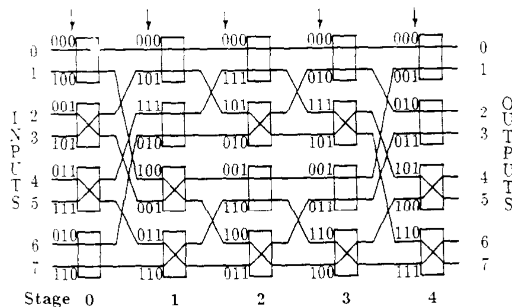

Fig. 4. Routing a linear permutation in $\mathcal{B}_3$ using Algorithm BL.

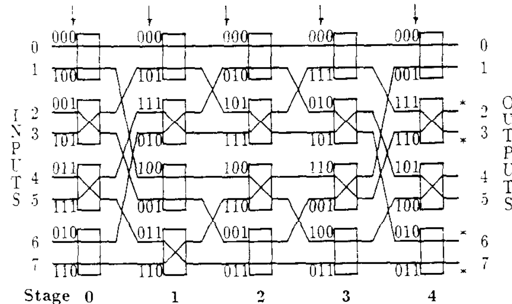

Fig. 5. The example linear permutation is incorrectly routed by Nassimi and Sahni's algorithm.

to the tag at the top input line, whereas Algorithm BL gives priority to the smaller tag.

A couple of examples of switch settings effected by Algorithm BL are shown in Fig. 3. For each switch, the destination tags at its inputs are shown, in binary form, and the routing bit is indicated by an arrow. In Fig. 3(a), the routing bit is 1 for both the tags; so there is a conflict in setting up the switch. This is resolved by comparing the destination tags and giving priority to the tag with smaller value, which, in this case, is at the lower input. The other tag is routed to the remaining output line. In Fig. 3(b), since the routing bits for two the tags are different each tag is routed according to its routing bit.

A complete example of this routing scheme is illustrated, using the linear permutation given by (5), in Fig. 4. Destination tags for each input line to a switch are given in the binary form. The routing bit for each stage is indicated by a downward arrow. Nassimi and Sahni's algorithm does not route this permutation. (See Fig. 5. Output ports with incorrect tags are marked with an asterisk.)

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}. \tag{5}$$

In stage 0, the routing bit is the same for both tags to a switch. Each switch in this stage is set up to route the smaller tag; for example, switch (0, 2) routes tag 011 to its lower output line as specified by the tag's 0th bit.

Due to its recursive construction, stages 1, 2, and 3 of $\mathcal{B}_3$ can be partitioned into top and bottom 4×4 Beneš networks, $\mathcal{B}_2$'s. (See Fig. 1.) The input addresses for these top and bottom $\mathcal{B}_2$'s are, respectively, $\{0, 2, 4, 6\}$ and $\{1, 3, 5, 7\}$. Notice that, after routing through stage 0, there exists a linear-complement permutation between y_2, y_1 of destination tags and x_2, x_1 of input lines for both the top and bottom $\mathcal{B}_2$'s given by equations, respectively, (6) and (7).

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{6}$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{7}$$

There exists conflict in setting up switches in stage 1 of the network as well. Switches 0 and 3 are set up such that the smaller tags, which are at the upper inputs, are routed, correctly; the other two switches are set such that the smaller tags, which are at the lower inputs, are routed, correctly. Switches in the last three stages are set up without conflicts.

B. An Algorithm for the $2n$-Stage Shuffle-Exchange Network

Algorithm PL: For the first n stages of $\prod_n$, switches are set up such that the destination tag with smaller value, when compared after bit reversal of the destination tags of both the inputs to the switch, is routed according to its routing bit. For the next n stages, switches are set up using the standard Ω self-routing algorithm. ■

The working of Algorithm PL is illustrated, using the example linear permutation given by (5), in Fig. 6. Routing bit at each stage is indicated by a downward arrow. This permutation is not realized by the $\prod_n$ self-routing algorithm of Yew and Lawrie [18]. Consider the tags 100 and 111 at switch (0, 1) in Fig. 6. They compete for the lower output line; since 100 is smaller, the switch is set route it correctly. For switch (0, 3), tag 110 (with bits reversed, has value 011) wins over tag 101 (with bits reversed, has value 101) and, hence, it is routed according to its routing bit (to the lower output line).

C. An Analysis of Algorithms BL and PL

First, we prove some results common to Algorithms BL and PL.

Consider a column (stage) of $N/2$ switches of size 2×2 with the ith bit as the routing bit; the upper input line to a switch has ith bit 0. At each input, there is a message with a destination address (tag). The relation between input line addresses and their tags is given by some permutation, $f \in \mathcal{LC}_n$. Then, f^{-1} is a linear-complement permutation giving the relation between the tags and input line addresses; it is expressed by a set of linear equations with each bit of x (line address) expressed as a linear combination of its y (tag) bits and the constant 1, if it is to be complemented.

Property 1: For any $\pi \in \mathcal{LC}_n$, let $\pi(x)$ denote the destination tag of any $x \in V$. Then, for any $a \in V$, $\pi(x)$ and $\pi(x \oplus a)$ differ in a bit if and only if, for all $x' \in V$, $\pi(x')$ and $\pi(x' \oplus a)$ differ in that bit.

For example, each pair of tags at the inputs of any switch, in the column described above, differ in the same bit positions. This can be readily shown using (2).

Let j be some bit position in which the destination tags to a switch differ. (If the destination tags to a switch differ in many bit positions, j could be any one of them.) Now, suppose the tags are routed through the column of switches using the following rule.

Routing rule: The destination tag with 0 as its jth bit is routed to the upper (respectively, lower) output line of the switch, if its ith bit is 0 (respectively, 1); the other tag is routed to the remaining output line. ■

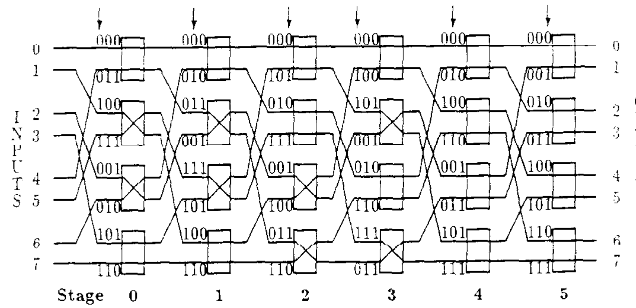

Fig. 6. Routing the example linear permutation in $\prod_3$ using Algorithm PL.

After routing, let g denote the relation from the line addresses to their corresponding tags, after routing. We prove the following results about g.

Lemma 1: The relation from line addresses to tags, after routing, is a linear-complement permutation.

Proof: It is clear that this relation, g, is a permutation. To prove the lemma, it is sufficient to show that g^{-1}, the permutation from tags to output line addresses, is a linear-complement permutation.

Depending upon how a switch is set, the line address at which the tag is present after routing is different from that before routing in at most one bit, namely, bit i. Therefore, irrespective of how the switches are set, the rth bit equation, $0 \leq r < n$ and $r \neq i$, of g^{-1} is the same as that of f^{-1}.

We need to show that x_i also is a linear combination of y-bits. There are two cases to consider.

Case 1 (no conflict): ith bits of the destination tags to a switch differ. Then the effect of routing is such that tags match their line addresses in bit i. That is, $x_i = y_i$.

Case 2 (conflict): ith bits of the destination tags to a switch are the same. Then, the tag with 0 in its jth bit is routed to the output line that matches it in the ith bit, and the other tag (whose jth bit is 1) is routed to the other output line, which does not match it in the ith bit. That is, $x_i = y_i \oplus y_j$.

In either case, the bit equation of x_i is as follows.

$$x_i = y_i \oplus \lambda y_j \tag{8}$$

where $\lambda = 0$ (for the "no conflict" case) or 1 (for the "conflict" case). ■

As an example consider the routing of the example linear permutation, (5), by Algorithm BL (same as the routing rule with i and j being, respectively, 0 and 2) through stage 0 of $\mathcal{B}_3$. (See Fig. 4.) Here, f^{-1} and g^{-1} are given by, respectively, (9) and (10). Notice that g^{-1} is different from f^{-1} in only the bit equation for x_0.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} \tag{9}$$

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}. \tag{10}$$

Lemma 2: Any two tags that differ in the ith bit only are routed such that one of them goes to the upper output line of some switch and the other goes to the lower output line of some switch.

Proof: Clearly, for the "no conflict" case, the lemma holds. We prove the validity of the lemma for the other case, by contradiction.

Let two tags s and t, which differ only in bit i, be routed to some two upper output lines. (The following argument is essentially the same when s and t are both routed to lower output lines.)

Any tag y routed to an upper output line satisfies the identity: $y_j \oplus y_i = x_i = 0$.

Since s and t are routed to upper output lines, $s_j \oplus s_i = t_j \oplus t_i = 0$. Noting that $s_j = t_j$, we get $s_i = t_i$, a contradiction. ■

Let $B = \{n-1, \cdots, 0\}$ be the set of all bit positions, and $B_i = \{n-1, \cdots, 0\} \backslash \{i\}$, the set of all bit positions except the bit position i. Also, let the relation from the set of line addresses with ith bit equal to 0 (respectively, 1) to their tags considering the bits in B_i be denoted g_0 (respectively, g_1). Then, g_0 and g_1 are said to be subrelations of g, when it is partitioned on bit i. In Lemma 3, we show that g_0, g_1 are indeed some permutations in $\mathcal{LC}_{n-1}$ and, hence, subpermutations of g, which is in $\mathcal{LC}_n$. For the above example, g is partitioned on bit 0 into g_0 and g_1 given by, respectively, (6) and (7).

Lemma 3: The relations g_0 and g_1 are some permutations in $\mathcal{LC}_{n-1}$.

Proof: From Lemma 2, any two tags routed to the upper lines differ in one or more bits other than the ith bit and, hence, are distinct considering only the bits in B_i. So g_0 is a permutation. Therefore, g_1 also is a permutation. To complete the proof, we show that $g_0^{-1}, g_1^{-1} \in \mathcal{LC}_{n-1}$.

From Lemma 1, after routing, the ith bit equation of g^{-1} is $x_i = y_i \oplus \lambda y_j$, where $\lambda \in \{0, 1\}$ [(8)]. The rth bit equation of g^{-1}, $0 \leq r < n$ and $r \neq i$, can be rewritten by substituting the occurrences of y_i, if any, with $x_i \oplus \lambda y_j$.

The rth bit equation of g_0^{-1} (g_1^{-1}) is obtained from that of g^{-1} by replacing any occurrence of x_i, which is a constant for the set of line addresses of this permutation, with its value—0 for the set of upper line addresses and 1 for the set of lower line addresses. That is, each bit equation of g_0^{-1} (g_1^{-1}) is free of x_i and y_i. ■

Observation 1: When expressed in the form of (2), g_0 and g_1 are given by the same $(n-1) \times (n-1)$ Boolean matrix, although they might differ in the "c" vector.

Corollary 2: Suppose a linear-complement permutation is to be routed through some k, $1 \leq k < n$, stages of switches according to the above routing rule using a distinct routing bit i_p in stage p, $1 \leq p \leq k$.

a) At any stage p, it is possible to select the "j" bit of the above routing rule, bit j_p in stage p, such that $j_p \notin \{i_1, \cdots, i_{p-1}\}$. (If $p = 1$, then $\{i_1, \cdots, i_{p-1}\}$ is the empty set.)

b) The relation from line addresses to tags after any stage p is a linear-complement permutation. Furthermore, it can be partitioned on bits $i_1, \cdots, i_p$ into 2^p subpermutations, each in $\mathcal{LC}_{n-p}$.

Proof: Both parts can be proved by induction using Lemma 3, for $k = 1$, as the basis. ■

All the results proved so far can be applied to Algorithms BL and PL due to the following claim.

Claim 4: a) The routing specified by Algorithm BL, in any of the first $n-1$ stages of $\mathcal{B}_n$, in realizing a linear-complement permutation

is the same as that given by the rule using the most significant bit (MSB), in which the tags to a switch differ, to decide the priority of an input line.

b) The routing specified by Algorithm PL, in any of the first n stages of $\prod_n$, in realizing a linear-complement permutation is the same as that given by the rule using the least significant bit (LSB), in which the tags to a switch differ, to decide the priority of an input line.

Proof is trivial.

Theorem 1: Algorithm BL routes any permutation of $\mathcal{LC}_n$ in $\mathcal{B}_n$.

Proof: Induction is used. It is trivially true for $n = 1$, since $\mathcal{B}_1$ is a 2×2 switch. For induction hypothesis, assume that any permutation in $\mathcal{LC}_m$, $m < n$, is realized correctly by Algorithm BL in a $\mathcal{B}_m$.

Consider the routing of a permutation $f \in \mathcal{LC}_n$ by Algorithm BL in $\mathcal{B}_n$. After routing through stage 0, the permutation g at the outputs of stage 0 switches can be partitioned on the 0th bit (the routing bit) into two subpermutations $g_0, g_1 \in \mathcal{LC}_{n-1}$. (Lemma 3.) Due to the inverse shuffle connection between stages 0 and 1, g_0 and g_1 appear at the inputs of, respectively, the top and bottom subnetworks $\mathcal{B}_{n-1}$'s, which comprise the stages $1, \cdots, 2n-3$ of $\mathcal{B}_n$. These subpermutations are correctly realized by Algorithm BL, in $\mathcal{B}_{n-1}$'s, by induction hypothesis. That is, after routing through stage $(2n-3)$, tags match their line addresses in bits $n-1, \cdots, 1$.

At the inputs of switches in stage $(2n-2)$, the last stage: the tags at a switch differ in only in the 0th bit, LSB, since the input line addresses to a switch differ only in LSB and each tag matches its line address in bits $n-1, \cdots, 1$. Hence, the switches in stage $(2n-2)$ can be set such that each tag is routed to the correct destination. ■

We now prove that Algorithm PL realizes linear-complement permutations correctly in $\prod$ networks.

Lemma 5: After routing a permutation in $\mathcal{LC}_n$ through the first n stages of $\prod_n$ using Algorithm PL, the relation between the line addresses and the tags is a lower triangular linear permutation.

Proof: First, we note that the relation between line addresses and tags after routing through a stage is some linear-complement permutation. (Corollary 2.) Let f_i denote the linear-complement permutation from line addresses to tags after routing through stage i, $0 \le i < n$. We show that f_{n-1}^{-1} (hence, f_{n-1}) is a lower triangular linear permutation.

From Lemmas 1 and 3, after routing through stage i, $0 \le i < n-1$, the equation of bit $n-1-i$ for permutation f_i^{-1} is of the form $x_{n-1-i} = y_{n-1-i} \dotplus \lambda_{n-1-i} y_j$, where $\lambda_{n-1-i} \in \{0, 1\}$ and $j < n-1-i$ (due to the comparison with *bits reversed*); furthermore, it is not changed later, since a bit equation in the permutation from tags to line addresses does not change, if that bit is not used for routing, and, of course, the ith bit is not used for routing again in the first half of $\prod_n$. After routing through stage $(n-2)$, the two tags at lines with addresses x and $x \dotplus 1$, for any $x \in V$, differ in the 0th bit, the only bit not yet used for routing. [Corollary 2a).] Since the tags at lines x and $x \dotplus 1$ meet at a switch in stage $(n-1)$, it can be set up without a conflict. Therefore, after routing through stage $(n-1)$, each tag matches its line address in the 0th bit. In summary, f_{n-1}^{-1} is of the following form.

$$\left.\begin{array}{ll} x_{n-1} = y_{n-1} \dotplus \lambda_{n-1} y_{j_{n-1}}, & j_{n-1} < n-1 \\ \vdots & \\ x_1 = y_1 \dotplus \lambda_1 y_{j_1}, & j_1 < 1 \\ x_0 = y_0 & \end{array}\right\}. \qquad (11)$$

It is clear that the above set of equations define a linear permutation with Boolean matrix in the lower triangular form. ■

For the routing of the example linear permutation of (5), the permutation from line addresses to tags after stage 2 of $\prod_3$, "f_{n-1}", is as follows. (See Fig. 6.)

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}.$$

Theorem 2: Algorithm PL routes any permutation of $\mathcal{LC}_n$ in $\prod_n$.

Proof: From Lemma 5, after the first n stages of routing by Algorithm PL, the resulting permutation is some lower triangular linear permutation, which can be routed in Ω_n, formed by the remaining n-stages of $\prod_n$. ■

Remarks:

1) Proof of Lemma 5 is very instructive in showing the effect of routing a linear-complement permutation by Algorithm PL in Ω_n. The effect of applying Algorithm BL to route a linear-complement permutation in Ω_n^{-1} is similar, except that the permutation from outputs to tags is an upper triangular linear permutation. (This can be shown readily by mimicking the proof of Lemma 5.) Thus, Algorithms BL and PL route linear-complement permutations by factorizing them into two simpler permutations, automatically. Lemma 5 shows the form of the "second" permutation, for the case of Algorithm PL. The "first" permutation is more complicated than the second one and is routed in the first half of the network. (This indeed is the approach discussed by Pease [13], although his treatment is not suitable for efficient routing.) Furthermore, after routing a given linear-complement permutation in Ω_n^{-1} (using Algorithm BL) or Ω_n (using Algorithm PL), the second permutation to be realized is simple enough to be realized by, for example, one of Steinberg's lower or upper triangular networks [15] or the second half of Waksman's network [17] (also, see Section IV-A).

2) Noting that $\mathcal{B}_n$ is simply Ω_n^{-1} followed by Ω_n with the last stage of Ω_n^{-1} identified with the first stage of Ω_n, an alternate proof for Theorem 1 can be given.

3) The effect of giving priority to the tag with larger value does not effect the correct routing of a linear-complement permutation by Algorithms BL and PL. However, the second permutation to be passed by the second half of the network is either an upper or a lower triangular linear-complement permutation. Now, the routing through a stage is described with the modified form of (8) of Lemma 1: $x_i = y_i \dotplus \lambda y_j \dotplus \lambda$.

4) If the bit used for resolving conflicts is already used in one of the earlier stages as routing bit, then the second permutation is neither an upper nor a lower triangular linear-complement permutation, but some linear-complement permutation simple enough to be passed by Ω_n or Ω_n^{-1} correctly. This can be observed, e.g., for Algorithm PL, by removing the restriction that $j_i < i, 0 < i < n-1$, in (11). Note that $x_0 = y_0$ is still true for this second permutation but, to show it, Corollary 2 has to be proved without the restriction that $j_p \notin \{i_1, \cdots, i_{p-1}\}$. It is readily shown by noting that the linear equations defining a permutation are independent.

The drawbacks of choosing an already used routing bit for conflict resolution are: a) the second permutation is slightly more complicated and b) it precludes on-the-fly switch setting for bit-serial networks (see Section IV-C).

IV. Other Applications of Algorithms BL and PL

A. Routing Linear-Complement Permutations in Waksman's Network

Waksman's network, $\mathcal{W}_n$, is simply a $\mathcal{B}_n$ with some switches ($\frac{N}{2} - 1$, to be precise) permanently set straight (hence, removed); this network is shown by Waksman to be rearrangeable with (asymptotically) the minimum number switches [17]. $\mathcal{W}_n$ is obtained from

$\mathcal{B}_n$ by setting the switches, given below, straight.

$$\left\{\text{switch}\left(n + i.2^{i+1}j\right) \mid 0 \leq i \leq n-2, 0 \leq j < N/2^{i+2}\right\}.$$

For example, $\mathcal{W}_3$ is obtained from $\mathcal{B}_3$ by setting the switches $\{(3,0), (3,2), (4,0)\}$ straight permanently.

Theorem 3: Algorithm BL routes linear-complement permutations in $\mathcal{W}_n$.

Proof: First, we investigate the setting of switches in $\mathcal{B}_n$ by Algorithm BL.

It is easily seen that tags with value 0 and 1 go to, respectively, upper and lower $\mathcal{B}_{n-1}$'s after routing through state 0. So, routing is completed with switch $(2n-2, 0)$ set straight. Since the subpermutations to be routed in the top and bottom $\mathcal{B}_{n-1}$'s are in $\mathcal{LC}_{n-1}$ (Theorem 1), the above argument can be recursively applied. So, the very switches always set straight by Algorithm BL in routing linear-complement permutations in $\mathcal{B}_n$ are the ones permanently set straight in $\mathcal{W}_n$. ■

B. An Algorithm for the $(2n-1)$-stage Shuffle-Exchange Network

Algorithm PL routes any $\pi \in \mathcal{LC}_n$ in $\prod_n$ with all the switches in the last stage are set straight, because $x_0 = y_0$ after the first n stages of routing [(11)]. See, for an example, Fig. 6. Therefore, we need only a $(2n-1)$-stage Shuffle-Exchange network followed by a perfect shuffle pattern (σ) to realize linear-complement permutations by Algorithm PL. Therefore, if we apply Algorithm PL to route π in $(2n-1)$-stage Shuffle-Exchange network, we actually route, $\sigma^{-1}\pi$, correctly, but not π. (Composition of permutations is right to left.)

For correct routing of π in $(2n-1)$-stage Shuffle-Exchange network, we modify Algorithm PL to treat destination tags as if a shuffle was performed on them; i.e., y_i is treated as $y_{i+1 \pmod n}$. With this modification, Algorithm PL actually attempts to route $\pi' = \sigma\pi$. And $(\sigma^{-1}\pi')$ is routed correctly, by Algorithm PL after $(2n-1)$ stages of Shuffle-Exchange. But, $\sigma^{-1}\pi' = \sigma^{-1}\sigma\pi = \pi$.

Theorem 4: With the above modification, Algorithm PL routes any permutation of $\mathcal{LC}_n$ in the $(2n-1)$-stage Shuffle-Exchange network.

Remark: The addresses of output lines of switches in the last stage of the $(2n-1)$-stage Shuffle-Exchange network do not match the network output port addresses. (This problem does not arise for $\mathcal{B}_n$ and $\prod_n$.) A perfect shuffle is required to match the output port addresses with the line addresses. This missing shuffle is compensated by treating the tags suitably. In general, if this "mismatch" is some $f \in \mathcal{BPC}_n$, then tags are processed with f before routing. If bit-controlled routing techniques are to be used, then f cannot be any other permutation.

C. Efficient Implementation of Algorithms BL and PL

Implementation of Algorithms BL and PL requires that the switches in the first half of the interconnection networks should be capable of comparing the tags. For the other half, since it is set using the Omega self-routing algorithm, extremely fast implementation techniques are known. In what follows, we discuss fast and efficient techniques for implementing the routing in the first half of the networks.

For word-parallel networks—each line of the network can carry one word at a time, this comparison operation represents additional time and hardware. However, with the current VLSI technology, comparison of two 32-bit words can be fast and, hence, the time penalty need not be severe. Since, with 32 bits, more than 4 billion lines can be indexed distinctly, for almost all networks of practical size, the routing overhead is manageable.

In bit-serial networks—each line in the network can carry one bit at a time, Algorithms BL and PL as stated appear inefficient, since switches in stage i, $1 \leq i < n-1$, need $(n-i)$ tag bits from the previous stage to set themselves. However, the following observation allows an efficient implementation of the algorithms with switches set on-the-fly. Thus, the switch setup times are similar to those with, for example, Nassimi and Sahni's algorithm [10].

Consider the effect of routing $\pi \in \mathcal{LC}_n$ by Algorithm PL in Ω_n. If there is a conflict in setting up switches in stage i, then $y_{j_{n-1-i}}$, where $j_{n-1-i} < n-1-i$ and is the LSB in which the two tags at a switch differ, is used to decide the priority [(11)]. However, y_p, where $p < n-1-i$ and is the MSB in which the two tags at a switch differ, could have been used for conflict resolution. (Of course, it is possible that $p = j_{n-1-i}$.) Even then, after routing through the first half, the second permutation to be passed through the second half of $\prod_n$ is still a lower triangular linear permutation, albeit a different one.

A similar observation holds for Algorithm BL. (Here, "pth bit" corresponds to the LSB not yet used for routing and the two tags at a switch differ in that bit.) Note that these modifications do not affect the results about the algorithms, proved earlier.

We organize tags as follows. For Beneš network, the actual tag consists of a $(2n-1)$-bit-stream sent through the network in sequence $0, \cdots, n-2, n-1, n-2, \cdots, 0$. (Message bits follow the tag bits.) For $\prod_n$, the tag consists of a $2n$-bit-stream sent through the network in sequence (left to right) $n-1, \cdots, 0, n-1, \cdots, 0$. At each input of the network, a tag organized as above followed by its message is fed, one bit at a time.

A switch in stage i, $0 \leq i < n$, of the network operates as follows. The first bit received is used as the routing bit. It examines the routing bits of its tags and removes them from their respective streams of bits. The remaining bits are sent to the next stage after setting itself as follows. It sets itself appropriately in the absence of conflicts and passes all the remaining bits it receives.

If there is a conflict, it sets itself straight or cross, depending on whether the routing bit value is 0 or 1, and send the incoming bit-streams to the outputs until they differ in a bit. (This is the "pth bit" of the above discussion.) If the bit at the upper input is 0, then the setting of the switch is not disturbed; otherwise, it is toggled—changed from cross to straight or vice versa. This bit and the following bits are sent to the switches in the next stage.

In the case of conflict, it does not matter how the switch is set, as long as both the incoming bits are the same. Once the streams differ in a bit, it can be set correctly. And conflict resolution is patterned after the above discussion. Hence, this implementation routes linear-complement permutations correctly.

D. Routing in Multistage Networks with Larger Size Switches

Algorithms BL and PL can be suitably modified to route linear-complement permutations in Beneš and $\prod$ networks constructed from $K \times K$ switches, $K = 2^k$ for $k \geq 1$. When N is not an integral power of K, switches in the middle stages can be of smaller size. An 8×8 Beneš network with 4×4 switches in the first and the last stages is shown in Fig. 7. We briefly sketch the modifications to the Algorithm BL, below. Similar modifications are applicable to Algorithm PL.

To set a $K \times K$, a group of k bits, which together form the routing digit, is used. In stage i, $0 \leq i < \lfloor \log_K(N) \rfloor$, bits $(i+1)k-1, \cdots, ik$ are used to form the routing digit. Conflicts in setting up a switch are resolved as follows. Apply the tags of a $K \times K$ switch to an Ω_k, Ω_k^{-1}, or some similar network made up of 2×2 switches with routing bits that form the routing digit of the switch; for conflict resolution, if necessary, complete tag may be used. (For a switch in the first stage of the network in Fig. 7, the network shown within it could be used with tag bits $0, 1$ for routing.) Route the tags by applying Algorithm BL or PL, whichever is appropriate, in this network. Now set the $K \times K$ switch such that

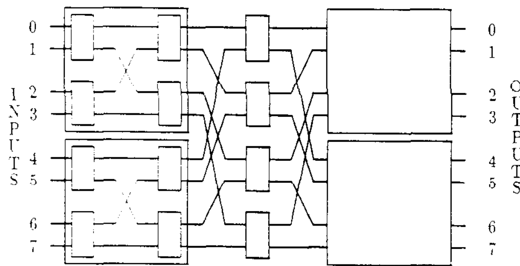

Fig. 7. A $\mathcal{B}_3$ with 4×4 switches in the first and last stages. The network within a switch in the first stage is used to determine its setting.

it simulates the routing specified by this $K \times K$ network built with 2×2 switches. There will not be any conflicts in routing through the remaining stages of the network.

V. Conclusions

In this paper, we have presented algorithms to route linear-complement permutations in Beneš, $\prod$, and $(2n - 1)$ stage Shuffle Exchange networks. The feature introduced by the algorithms is: when there are conflicts in setting up a switch, one of the tags is given priority based on some type of comparison operation. When there are no conflicts in setting up a switch, the routing is similar to the Omega self-routing algorithm. These algorithms can be efficiently implemented in bit-serial networks. For word-parallel networks, the overhead of comparison operation is reasonable for networks with, say up to 4 billion inputs and outputs.

Algorithm BL realizes the class of inverse Omega permutations in Beneš networks, since there will not be any conflicts in setting up the switches. Similarly, Algorithm PL realizes any Ω permutation in $2n$-stage Shuffle-Exchange networks. In fact, the classes of permutations realized by these algorithms are much larger than the linear class. It is interesting to note that Algorithm BL routes all permutations in $\mathcal{B}_2$. However, it does not route all Omega permutations in larger size Beneš networks. If a permutation is known to be in Omega class, then it can be realized in a Beneš network by setting the first $(n - 1)$ stages of the network straight, as suggested by Nassimi and Sahni [10].

Using the analysis techniques presented, we have shown that the following classes of permutations are realized by Algorithm BL on $\mathcal{B}_n$ [14]. Similar results are shown for Algorithm PL. To pass these permutations, conflict resolution should be as given for the case of bit-serial networks.

1) Any permutation of the form πv, where v is an Omega admissible upper triangular permutation [15] and $\pi \subset \mathcal{LC}_n$. These types of permutations are useful in parallel memory access of sub- and superdiagonals of data matrices [4].

2) Any permutation partitionable on the first k least significant bits into 2^k subpermutations—each may be distinct and similarly partitioned. Permutations of this type are useful in partitionable SIMD systems.

Further work in characterizing the classes of permutations realized by the proposed algorithms is needed. Another direction for further work is in developing simple control algorithms to route any permutation, specified as an algebraic function, since such types of permutations are used frequently in parallel processing.

References

[1] J. Beetem, M. Denneau, and D. Weingarten, "The GF11 supercomputer," in *Proc. Int. Symp. Comput. Architecture*, 1985, pp. 108–115.

[2] V. E. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic.* New York: Academic, 1965.

[3] G. Birhkoff and S. MacLane, *A Survey of Modern Algebra*, fourth ed. New York: Macmillan, 1977.

[4] R. V. Boppana and C. S. Raghavendra, "Generalized schemes for access and alignment of data in parallel processors with self-routing interconnection networks," *J. Parallel Distributed Comput.*, vol. 11, pp. 97–111, 1991.

[5] T. Etzion and A. Lempel, "An efficient algorithm for generating linear transformations in a shuffle exchange network," *SIAM J. Comput.*, vol. 15, no. 1, 1986.

[6] K. Hoffman and R. Kunze, *Linear Algebra*, second ed. Englewood Cliffs, NJ: Prentice-Hall, 1971.

[7] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, no. 12, 1975.

[8] J. Lenfant, "Parallel permutations of data: A Beneš network control algorithm for frequently used permutations," *IEEE Trans. Comput.*, vol. C-27, 1978.

[9] D. Nassimi, "A fault-tolerant routing algorithm for BPC permutations on multistage interconnection networks," in *Proc. Int. Conf. Parallel Processing*, 1989, pp. 278–287.

[10] D. Nassimi and S. Sahni, "A self-routing Beneš network and parallel permutation algorithms," *IEEE Trans. Comput.*, vol. C-30, no. 5, 1981.

[11] ——, "Parallel algorithms to set up the Beneš permutation network," *IEEE Trans. Comput.*, vol. C-31, pp. 148–154, 1982.

[12] D. S. Parker, "Notes on shuffle/exchange-type switching networks," *IEEE Trans. Comput.*, vol. C-29, pp. 213–222, 1980.

[13] M. C. Pease, III, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, 1977.

[14] C. S. Raghavendra and R. V. Boppana, "An analysis of some self-routing schemes for multi-stage interconnection networks," Tech. Rep., Dep. EE–Systems, Univ. of Southern Cal., Univ. Park, Los Angeles, CA 90089-0781, June 1990.

[15] D. Steinberg, "Invariant properties of the shuffle-exchange and a simplified cost-effective version of the omega network," *IEEE Trans. Comput.*, vol. C-32, pp. 444–450, 1983.

[16] A. Varma and C. S. Raghavendra, "Rearrangeability of multistage shuffle/exchange networks," *IEEE Trans. Commun.*, vol. COM-36, no. 10, 1988.

[17] A. Waksman, "A permutation network," *J. ACM*, vol. 15, no. 1, 1968.

[18] P.-C. Yew and D. H. Lawrie, "An easily controlled network for frequently used permutations," *IEEE Trans. Comput.*, vol. C-30, no. 4, 1981.