A SELF–ROUTING PERMUTATION NETWORK[†]

DAVID M. KOPPELMAN
2735 East 65th Street
Brooklyn, NY 11234

A. YAVUZ ORUÇ
Electrical Engineering Department
University of Maryland
and
Institute for Advanced Computer Studies
College Park, MD 20742

ABSTRACT

A self–routing permutation network is a connector which can set its own switches to realize any one-to-one mapping of its inputs onto its outputs. Many permutation networks have been reported in the literature, but none with self–routing property, except crossbars and cellular permutation arrays which have excessive cost. This paper describes a self–routing permutation network which has $O(\log^3 n)$ bit-level delay and uses $O(n \log^3 n)$ bit-level hardware where n is the number of inputs to the network. The network is derived from a complementary Beneš network by replacing each of its two switches in its first stage by what is called a *1–sorter*, and recursively defining the switches in the third stage as self–routing networks. The use of 1–sorters results in substantial reduction in both propagation delay and hardware cost when contrasted with $O(n)$ delay and $O(n^{1.59})$ hardware of the recursively decomposed version of a complementary Beneš network. Furthermore, these complexities match the propagation delay and hardware cost of Batcher's sorters (only networks, other than crossbars and cellular permutation arrays, which are known to behave like self–routing permutation networks.) More specifically, it is shown that the network of this paper uses about half of the hardware with about four-thirds of the delay of a Batcher's sorter.

[†] This work is supported in part by the National Science Foundation under grant No:CCR-8708864

I. Introduction

A network with n inputs and n outputs and one which can realize all permutations of its inputs onto its outputs is called a *permutation network* and finds applications as a connector in many switching systems [2,5,14,15] and parallel computers [6,7,25,26]. If, in addition, such a network can connect its inputs to its outputs by decoding the destinations of the inputs switch by switch rather than by using a global routing procedure, it is then called a *self–routing* permutation network. The advantage of a self–routing scheme over a global one is that the routing time of a self–routing network matches its propagation delay. Furthermore, if the address decoding logic at each switch can be kept simple then the hardware cost of a self–routing network will, in general, be less than that of a network with a global routing scheme. Consequently, self–routing permutation networks can enhance the connection power of $\log n$-stage[1] shuffle-exchange networks without exacting any time penalty, unlike other permutation networks which need $O(n \log n)$ time to set up [19,28].

An ordinary self–routing permutation network is a "crossbar" switch which is just an $n \times n$ matrix of crosspoints. Suppose that the inputs enter at the rows and the outputs exit vertically through the columns. Then a self–routing scheme can be implemented by a simple *decrement and test for zero scheme* as follows: if input i is to be routed to output j then the destination of i is set to j and is decremented every time it moves to the next crosspoint on the ith row until j becomes zero. Once it reaches the jth column, it is then projected down along this column until it reaches output j. Obviously, no conflicts can arise in simultaneously routing inputs to outputs as long as each input is routed to at most one output, and each output is reached from at most one input. This is, definitely, the case for any one-to-one maps including permutations, hence a crossbar switch can self-route permutations. Like crossbars, cellular permutation arrays [8,22] can also be made self–routing by using simple address decoding techniques [11,21].

Even though crossbars and cellular permutation arrays are self–routing, they are expensive networks as compared to other permutation networks. One way to alleviate the cost problem is to use an asymptotically optimal permutation network [2]. However, the fastest routing algorithm for this network runs in $O(\log^2 n)$ time on a parallel machine with a more complex interconnection structure than the network itself [13,17]. A few efforts in the literature point out that the Benes network can be self–routed for some permutations[3,18], but no self–routing scheme which works uniformly over all permutations has ever been

[1] All logarithms in this paper are in base 2 and it is further assumed that n is a power of 2 unless otherwise stated.

reported.

Another alternative for self–routing is to use a sorting network. Sorting is closely related to permuting since if one can sort a set of items into an ascending (or descending) order, one can also permute them simply by sorting their addresses. Self–routing networks which use sorting techniques have been reported in the literature [9]. The most widely-known of these are Batcher's sorting networks [1,10] which are based on odd-even and bitonic merging schemes. Both even-odd and bitonic sorters will be referred to as *Batcher's networks* as they have the same hardware and time complexity; their gate–level hardware complexity is $O(n \log^3 n)$ and propagation delay is $O(\log^3 n)$.

Another way of viewing a self–routing permutation network is as a connector whose programming hardware is interspersed with its data path. Using this point of view, this paper will describe a self–routing permutation network which has the same hardware cost and propagation time complexity as those of Batcher's networks; but which also has features of its own. First, this network is not made up of compare/swap units like Batcher's networks, and therefore, is not a standard sorting network. Rather, it is derived from a particular Clos network [5], called the complementary Benes network [4] by replacing its two large input-stage switches with devices called *1–sorters*, and each of its two output-stage switches by a self–routing permutation network. Second, while its cost and propagation delay are asymptotically identical to those of a Batcher's network, its cost is about one-half of the cost, and its propagation delay is about four-thirds of the delay of the same network by a conservative estimate. Finally, this new network is not a standard self–routing connector in that its switches use ranking to decide their outputs, while in a standard self–routing connector, such as an omega network [12], the switches are set strictly by examining the bits of destination addresses.

The remainder of the paper will describe this self–routing permutation network in detail. In Section II a brief description of the complementary Beneš network is given. The self–routing network and 1–sorter are presented in Sections III and IV respectively. In Section V, the cost and propagation delay of this network are derived and contrasted with the cost and propagation delay of a Batcher's network. The paper is concluded in Section VI.

II. The Complementary Beneš Network

The underlying structure behind the self–routing network is the complementary Beneš network (CBN) which is depicted in Figure 1. The network consists of three stages, an input stage (the first column of two switches), a center stage of $n/2$ 2-input switches, and an output stage (the last column of two switches.) The inputs are listed on the left hand

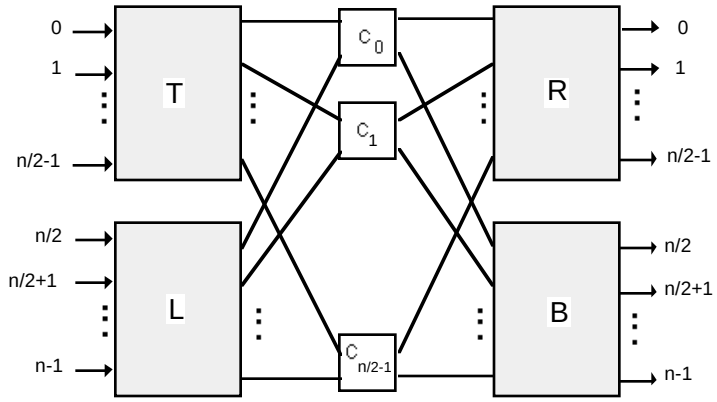

Figure 1. A complementary Beneš network.

side and the outputs are listed on the right hand side. The 2-input switches in the center stage are crossbars; their inputs either go straight through to their outputs (called the *identity* connection), or the top input goes to the bottom output, and the bottom input goes to the top output (called the *transpose* connection).

The top switch in the input stage is denoted T, the bottom switch is denoted L; the top switch in the output stage is denoted R, and the bottom switch is denoted B. The 2-input crossbars in the center are labelled $C_0, C_1, \ldots, C_{n/2-1}$. Each of the large switches has $n/2$ outputs, the inputs and outputs of T and R are labelled $0, 1, \ldots, n/2-1$, and the inputs and outputs of L and B are labelled $n/2, n/2+1, \ldots, n-1$. C_i's inputs are connected to the ith output of T and ith output of L and C_i's outputs are connected to the ith input of R and ith input of B for $0 \leq i \leq n/2-1$.

It is known that an n-input CBN can permute its inputs onto its outputs in any one of $n!$ ways [2,5]. Furthermore, it is also known that one of the input (or output)-stage switches can be removed or permanently set to a given state without destroying network's permutation capability [20,23]. However, the self–routing version of CBN requires that all four switches in the input and output stages be present as described in the next section.

III. The Self–routing Network

The self–routing network appears in Figure 2. Just as the CBN, the self–routing network consists of an input stage, a center stage, and an output stage. The input stage consists of two $n/2$-input devices, called 1–sorters and denoted T and L , the center stage consists of

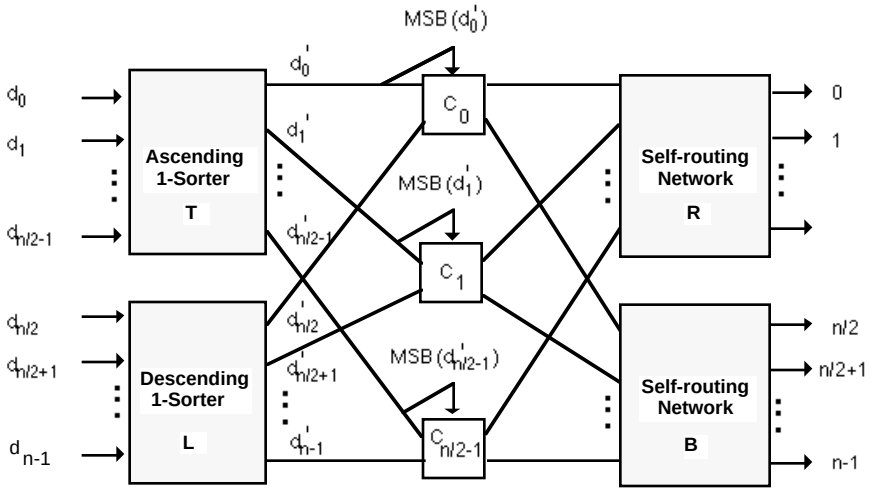

Figure 2. The self–routing permutation network.

$n/2$ 2-input crossbars, and the output stage consists of two $n/2$-input self–routing networks, denoted R and B. Each of the n inputs to the network consists of w-bit data word and d-bit destination address, where $d = \log n$. For clarity, only the destination addresses are shown in Figure 2. The destination address d_i specifies the output on the right hand side to which the w-bit data word on input i is to be routed. Each input to the network enters one of the two 1–sorters. The upper 1–sorter in the first stage "sorts" its inputs onto its outputs in ascending order, using the most significant bit of each of the destination addresses as the sort keys. The sorted outputs are indicated as $d'_i; 0 \leq i \leq n/2 - 1$ in Figure 2. The lower 1–sorter works similarly, except that it sorts its inputs in descending order. One can easily form a descending 1–sorter from an ascending 1–sorter simply by inverting the most significant bits of the sort keys. Rather than carrying out this step explicitly on the sort keys of the lower 1–sorter, we shall use both types of 1–sorter in our discussion for clarity.

The following establishes that this structure forms a self–routing permutation network.

Theorem 1: The network in Figure 2 can realize any permutation of its inputs onto its outputs on a self–routing basis.

Proof: We first note that each center–stage switch receives one input from 1–sorter T, and one input from 1–sorter B in the first stage. Suppose that, under a given permutation p, x of the inputs of T are mapped to some x outputs of self–routing network R and $n/2 - x$ are mapped to some $n/2 - x$ outputs of self–routing network B for some integer $x; 0 \leq x \leq n/2$. Since T sorts its inputs in ascending order using the most significant bits

of the addresses of those inputs as keys, all inputs of T which are destined to the outputs of R are routed to the upper inputs of the first x switches in the center stage, starting at the top. Furthermore, the inputs for the remaining $n/2 - x$ outputs of R must be drawn from the inputs of 1–sorter L in the first stage. This in turn implies that x of the inputs of L will be mapped to some x outputs of B. Now, since L sorts its inputs in descending order, these x inputs will end up going to the lower inputs of the top most x switches in the center, thereby pairing up with the other x inputs which arrive at the upper inputs of these same x switches from T, and are destined to the outputs of R. Consequently, each of these x switches will have one input which is to be mapped to the outputs of R and one input which is to be mapped to the outputs of B. Similarly, the $n/2 - x$ inputs T which are destined to the outputs of B are paired up with the $n/2 - x$ inputs of L which are destined to the outputs of R. Hence, the remaining $n/2 - x$ switches will also receive one input from each of T and L in the first stage and map one to an output of R and the other to an output of B.

As for setting the switches in the center, the following simple self–routing scheme suffices: If the most significant bit of the address emerging from the ith output of T is 1 then switch C_i is set to the transpose state, and if it is 0 then C_i is set to the identity state. It is obvious that this guarantees that each of the inputs arriving at that switch from T and L is routed to the correct subnetwork in the third stage. Since networks R and B are also self–routing, the entire network can self–route permutation p. Furthermore, this is true for all $x; 0 \leq x \leq n/2$ and any permutation p of the network's inputs onto its outputs, and so the statement follows. ||

In the network of Figure 2 each of the self–routing networks can further be decomposed into a 3-stage self–routing network by using two 1–sorters, each with $n/4$-inputs, a set of $n/4$ 2-input switches and two self–routing networks, each with $n/4$ inputs. This process can be repeated until each of the networks in the output stage becomes trivial. Figure 3 depicts an 8-input self–routing network which is obtained recursively by decomposing the self–routing networks in the third stage of the original self–routing permutation network. Self–routing permutation networks with larger numbers of inputs can be constructed similarly.

Figure 4 illustrates how a permutation is routed through an 8-input self–routing network. As indicated by the captions, the light gray boxes represent ascending 1–sorters, the dark ones represent descending 1–sorters, and the remaining boxes are 2-input switches. The pairs of symbols on the left denote the data and their destination addresses; for example, datum w_0 is to be routed to output 5, w_1 to output 2, and so on. The paths are established as described in Theorem 1. For example, the four inputs to the upper 1–sorter in the first

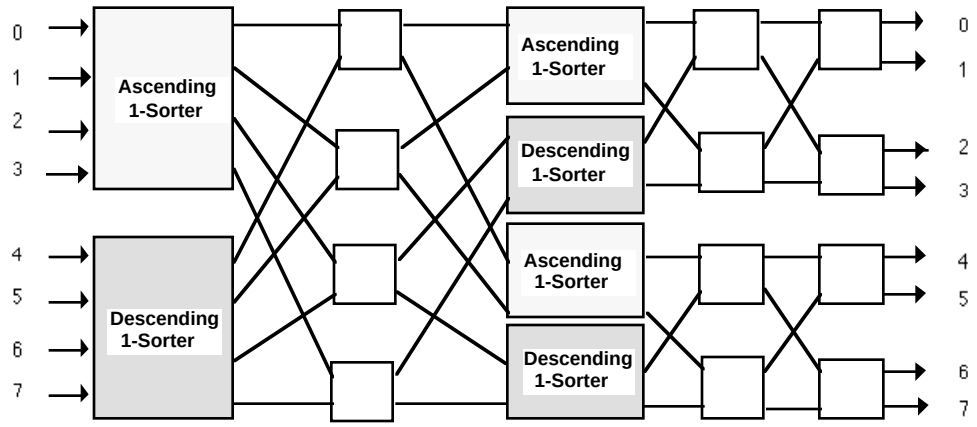

Figure 3. 8-input self–routing network.

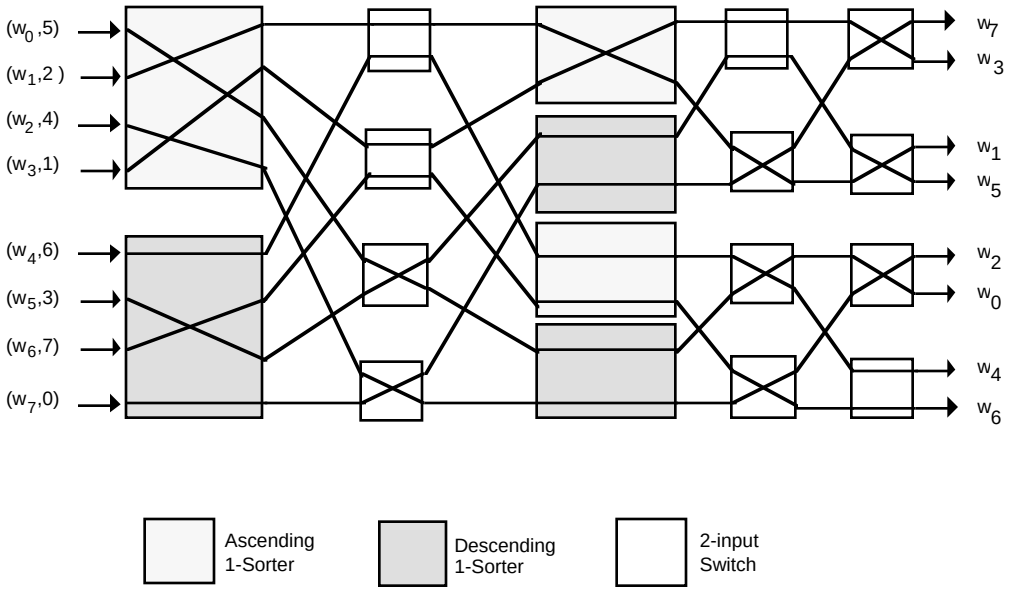

Figure 4. Illustration of self–routing.

stage are sorted by using the most significant bits of destinations 5,2,4 and 1. Thus, any one of the four sequences

(2,1,5,4),(1,2,5,4),(2,1,4,5),(1,2,4,5)

is an ascending output sequence. Note that the emphasis is on whether the most significant bits of the addresses are 0 or 1 (0 for inputs 1 and 2, and 1 for inputs 5 and 4). A 2-input switch is set to the transpose state if the most significant bit of the address at its upper

input is 1, and to the identity state if it is 0. Thus, the top two switches in the second stage in the network of Figure 4 are set to the identity while the bottom two are set to the transpose state. The other switches are similarly set.

IV. The 1–sorter

In order to complete the design of the self–routing network described in Section III, we must provide an explicit construction of a 1–sorter using elementary devices. One way to implement a 1–sorter is to *rank* its inputs by their keys and then *concentrate* the inputs by their ranks. In what follows we shall use this approach to design a descending 1–sorter. An ascending 1–sorter can be obtained from a descending one by inverting its inputs.

First, we need a few definitions.

Definition 1: Let $k_i; 0 \leq i \leq n-1$ be a sequence of 1-bit keys. The *rank*, r_0 of key k_0 is -1 if $k_0 = 0$ and 0 if $k_0 = 1$. The rank, r_i of key $k_i; i \geq 1$ is r_{i-1} if $k_i = 0$, and $r_{i-1} + 1$ if $k_i = 1$. For example, if the keys were 0,1,1,0,1,0, the ranks would be -1,0,1,1,2,2. ||

Definition 2: A *ranking circuit* of size n is a device with n inputs and n outputs, which returns the rank of its ith input on its ith output. (Input i has rank r_i if its key has rank r_i. ||

Definition 3: An $n \times n$ concentrator is a connector which can map any subset of its inputs onto a subset of consecutive outputs of equal cardinality. If this mapping can be done by decoding the destination addresses of the inputs switch by switch then the concentrator will be called a *self-concentrator*. ||

Figure 5 depicts how to combine a ranking circuit with a concentrator to obtain a 1–sorter. The ranking circuit receives the most significant bits of the destination addresses $d_i; 0 \leq i \leq n-1$, and returns their ranks at its outputs. The ranks are then concatenated with the data inputs (i.e., $w_i; 0 \leq i \leq n-1$) and the destination addresses, and fed into a self-concentrator shown on the right in the figure. Using the ranks of the inputs, the concentrator then maps those inputs whose destination addresses carry a 1 in their most significant bits to a set of consecutive outputs starting with the top most output. The remaining inputs, i.e., those inputs whose destination addresses carry a 0 in their most significant bits are automatically routed to the remaining outputs. Thus the schematic in Figure 5 1–sorts its inputs in descending order.

Ranking Circuit

A ranking circuit of n inputs can be formed recursively from two ranking circuits, each with $n/2$ inputs, and $n/2$ 2-input adders with an additional 1-bit input which is set to 1

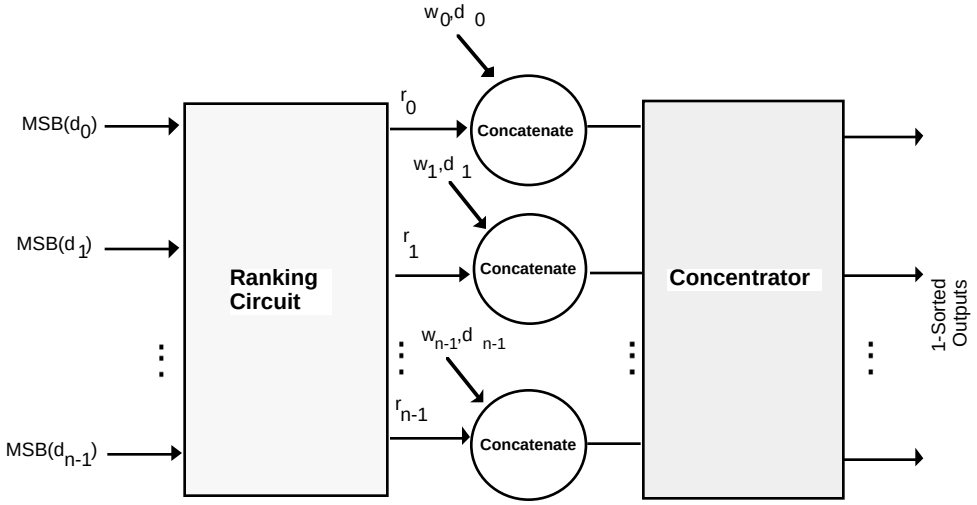

Figure 5. A concentrator based 1–sorter.

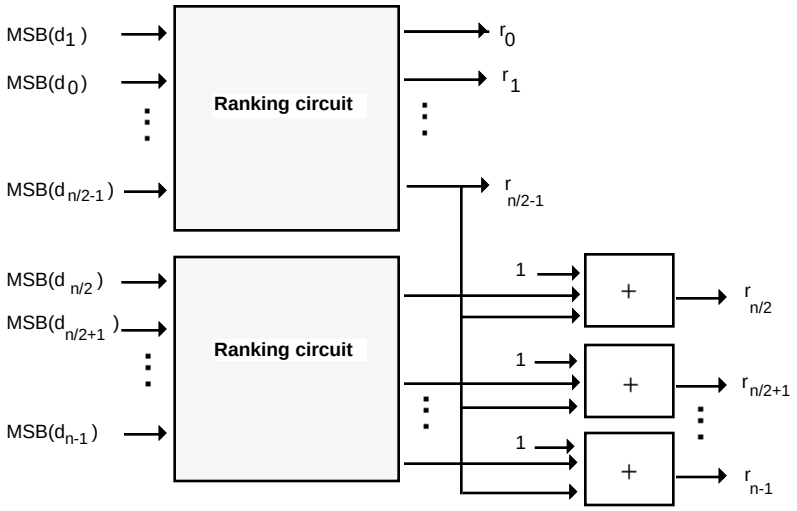

Figure 6. A recursive construction of a ranking circuit by using elementary adders.

as shown in Figure 6. The ranks of the first $n/2$ inputs are collected directly from the outputs of the upper ranking circuit, and the ranks of the bottom half inputs are obtained by adding the rank of input $d_{n/2-1}$ to the outputs of the bottom ranking circuit. By decomposing each of the $n/2$-input ranking circuits recursively, one can then form a $\log n$-depth ranking circuit consisting of $1 + \frac{n}{2}\log n$ 2-input adders. Such a ranking circuit is shown for $n = 8$ in Figure 7.

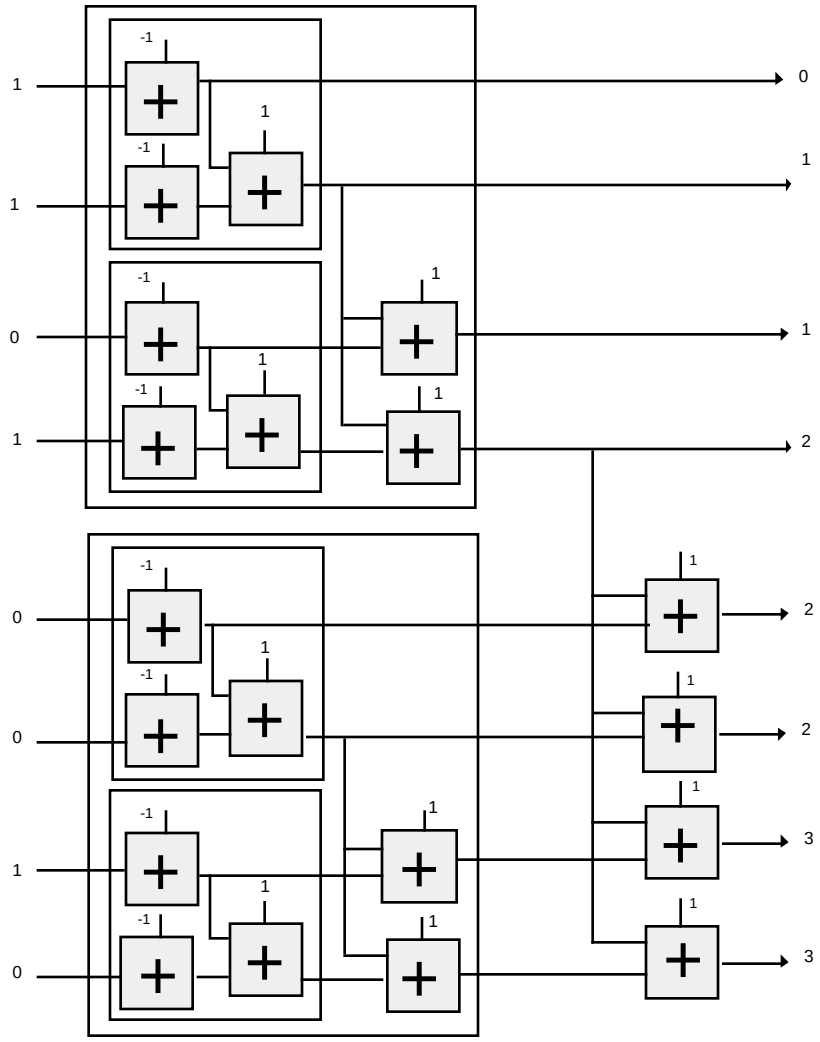

Figure 7. An 8-input ranking circuit.

A more economical ranking circuit, one which uses $O(n)$ 2-input adders, and of depth $O(\log n)$ can be obtained by using a tree of depth $O(\log n)$. To facilitate this, we first rank the input keys slightly differently than as described in Definition 1. The ranks of 1 keys are left unchanged, but the rank of each 0 which immediately follows a 1 is incremented by one. For example, for the sequence given in Definition 1, i.e., 0,1,1,0,1,0, the corresponding new ranks are 0,0,1,2,2,3. We note that this change does not cause any problem since we only need the ranks of 1 keys in designing a 1–sorter and those ranks remain intact.

Now, each node in the tree is a simple combinational unit—to be called a Y–unit—which can be one of four devices as shown in Figure 8: (1) a root Y–unit which is at the root of

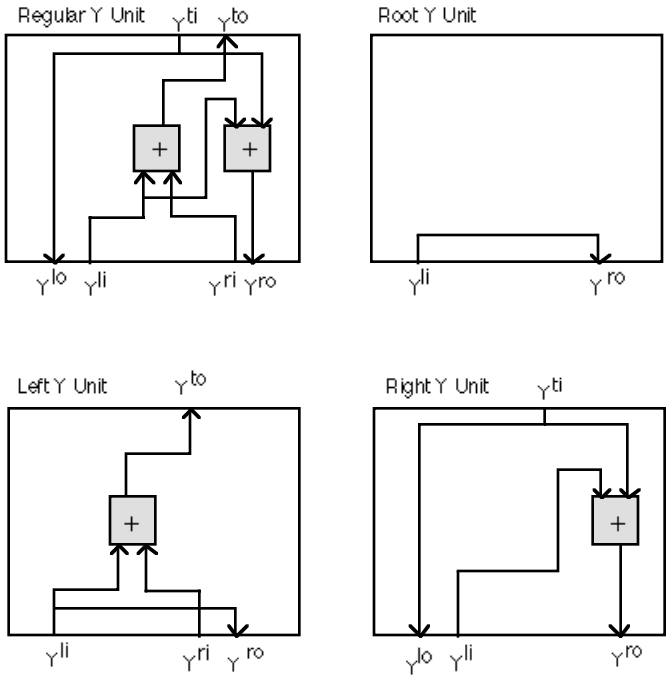

Figure 8. Y–units.

the tree, (2) left Y–units which are the nodes between the leftmost leaf and the root, (3) right Y–units which are the nodes between the rightmost leaf and the root, and (4) regular Y–units which are the remaining nodes. The Y–units have three sets of inputs and three sets of outputs; only the regular Y–units have all of these connections; which connections the root, left, and right Y–units have are described below along with their function.

The input to a regular Y–unit entering from its parent node is a $\log n$-bit word, denoted Y^{ti}, the outputs to the child nodes are $\log n$-bit words called Y^{lo} and Y^{ro}. The inputs to a Y–unit from the child nodes are also $\log n$-bit words called Y^{li} and Y^{ri}. The output to the parent node is a $\log n$-bit word called Y^{to}. A left Y–unit has all but terminals Y^{ti} and Y^{lo}, and a right Y–unit has all but terminals Y^{ri} and Y^{to} of a regular Y–unit.

There are $n/2$ leaf nodes in the tree (assuming an even n); call the input vector of 0's and 1's to be ranked $k_0, k_1, \ldots k_{n-1}$. Then the input vector is connected to the leaves in the following way: k_0 and k_1 are connected to the Y^{li} and Y^{ri} inputs of the left most leaf in that order, k_2 and k_3 are connected to the Y^{li} and Y^{ri} inputs of the next left most leaf in that order, and so on up to k_{n-2} and k_{n-1} which are connected, respectively, to the Y^{li} and Y^{ri} inputs of the rightmost leaf.

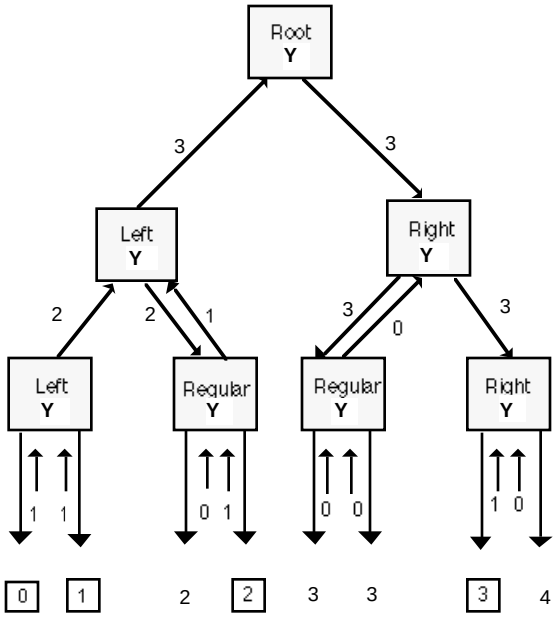

Figure 9. An 8-input ranking circuit implemented with Y–units.

A regular Y–unit realizes the following function:

$$\mathrm{Y}^{to} = \mathrm{Y}^{li} + \mathrm{Y}^{ri}, \mathrm{Y}^{lo} = \mathrm{Y}^{ti}, \mathrm{Y}^{ro} = \mathrm{Y}^{li} + \mathrm{Y}^{ti};$$

a left Y–unit realizes the following function:

$$\mathrm{Y}^{to} = \mathrm{Y}^{li} + \mathrm{Y}^{ri}, \mathrm{Y}^{ro} = \mathrm{Y}^{li};$$

a right Y–unit realizes the following function:

$$\mathrm{Y}^{lo} = \mathrm{Y}^{ti}, \mathrm{Y}^{ro} = \mathrm{Y}^{li} + \mathrm{Y}^{ti};$$

and the root Y–unit realizes the trivial function

$$\mathrm{Y}^{ro} = \mathrm{Y}^{li}.$$

The leftmost leaf Y–unit is special in that it has a Y^{lo} output which is permanently set to 0. Similarly, the rightmost leaf Y–unit is special in that it receives three inputs but only those arriving at Y^{ti}, and Y^{li} are used in computing the rank of its inputs.

To see how the Y–units compute the rank of the 1's at the leaves, refer to Figure 9. The 1's and 0's enter the tree at the leaves. Each leaf adds its lower inputs and sends the sum up to its parent. The data moving up the tree along the path from the leftmost node to the root can all be interpreted as the rank of the next 1 to be encountered. A datum moving up the tree elsewhere, can be interpreted as the number of 1's below that datum. These data continue going up the tree until they encounter a node which is part of the path from the root to the left most node. A datum going down the tree along the path from the root to the rightmost node can be interpreted as the rank of the next 1 to be encountered. Note that, 1 and 0 inputs are both assigned ranks. However, as stated before, the ranks of 0 inputs will be ignored when inputs are self–routed through the concentrator stage, and only the ranks of 1 inputs will be used.

Self-concentrator

Once the ranking of the keys is completed, the ranks along with the destinations and data are fed to a concentrator as shown in Figure 5. Suppose that the concentrator's outputs are numbered $0, 1, \ldots, n-1$ from top to bottom, and also suppose that the destination addresses of exactly x of the inputs to the concentrator have their most significant bits set to 1. The concentrator then routes those x inputs to its lowest numbered x outputs, and its remaining inputs to its remaining outputs.

This kind of concentrator device can be implemented by what is widely known as a multistage cube network [24,29]. It was established [16,27] that such a network concentrates any subset of its inputs to any subset of consecutive outputs of equal cardinality. It is also known that the same network can self–route any of its inputs to any of its outputs. What needs to be shown is whether this network is also self–concentrating as specified in Definition 2. The procedure given below shows that the network indeed self-concentrates if the ranks of the destinations of its inputs are known.

The procedure is straightforward: The switches are set by the ranks and destinations of inputs as they traverse through the network. Let S_i denote an arbitrary but fixed switch in the ith stage of an n-input cube network where the stages are numbered $0, 1, \ldots, \log n - 1$ from left to right. Let $d_{u,\log n-1} d_{u,\log n-2}, \ldots, d_{u,0}$ be the binary representation of the destination address and let $r_{u,\log n-1}, r_{u,\log n-2}, \ldots, r_{u,0}$ be the binary representation of the rank which arrive at the upper input of S_i. Similarly, let $d_{l,\log n-1} d_{l,\log n-2}, \ldots, d_{l,0}$ and $r_{l,\log n-1}, r_{l,\log n-2}, \ldots, r_{l,0}$ denote in binary, respectively, the destination address and the rank which arrive at the lower input of switch S_i. Switch S_i is set as shown in Table 1.

Table 1

$r_{u,i}, r_{l,i}$	$d_{u,0}, d_{l,0} = 00$	$d_{u,0}, d_{l,0} = 01$	$d_{u,0}, d_{l,0} = 10$	$d_{u,0}, d_{l,0} = 11$
00	–	*transpose*	*identity*	*impossible*
01	–	*identity*	*identity*	*identity*
10	–	*transpose*	*transpose*	*transpose*
11	–	*identity*	*transpose*	*impossible*

As seen from the table, the switches in stage $i; 0 \leq i \leq \log n - 1$ are set by using the most significant bits of the destination addresses and the ith bits of the ranks they receive. The ranks act very much like the destination addresses in an ordinary cube network, while the most significant bits of the destination addresses, i.e., bits $d_{u,0}$ and $d_{l,0}$, determine the way that the ranks set the switches.

More precisely, when $d_{u,0} = d_{l,0} = 0$, the ranks which arrive at switch S_i are ignored and S_i can be set to either the identity or transpose state as indicated by the $--$ entries in the second column. This is in keeping with the fact that the network is to concentrate only those inputs whose addresses have a 1 in their most significant bits.

If $d_{u,0} = 0$ and $d_{l,0} = 1$ then S_i is set using the ith bit of the rank which arrives at the lower input of S_i. On the other hand, if $d_{u,0} = 1$ and $d_{l,0} = 0$ then S_i is set using the ith bit of the rank which arrives at the upper input of S_i. These cases are shown in the third and fourth columns of the table. Again, these settings are in agreement with the fact that the network is to concentrate only those inputs whose addresses carry a 1 in their most significant bits.

Finally, if $d_{u,0} = 1$ and $d_{l,0} = 1$ as shown in the fifth column, then there are two cases to consider: The first is that the ranks which arrive at S_i differ in the ith bit position. In this case, there is no conflict and the ranks are used as ordinary destination addresses as in the earlier cases to set S_i. The second case is when the ranks are identical in the ith bit position, i.e., their ith bits are both 0, or both 1. Obviously, if this happens then the network cannot self-concentrate, but we can show that it never does, as follows. First, for $i = 0$ it is obvious that the ranks which arrive at any switch S_i in stage 0 must differ in bit position 0 when $d_{u,0}$ and $d_{l,0}$ are both 1. Suppose that, for a given $i; 1 \leq i \leq \log n - 1$, some two ranks, R_x and R_y arrive at a switch S_i in stage i. This implies that R_x and R_y must have originated from some two inputs which are tied to a cube network of 2^i inputs which is contained in the original n-input cube network. Moreover, the bits $i-1, i-2, \ldots, 0$ of R_x and R_y must be identical, since otherwise, they cannot not both be routed to the

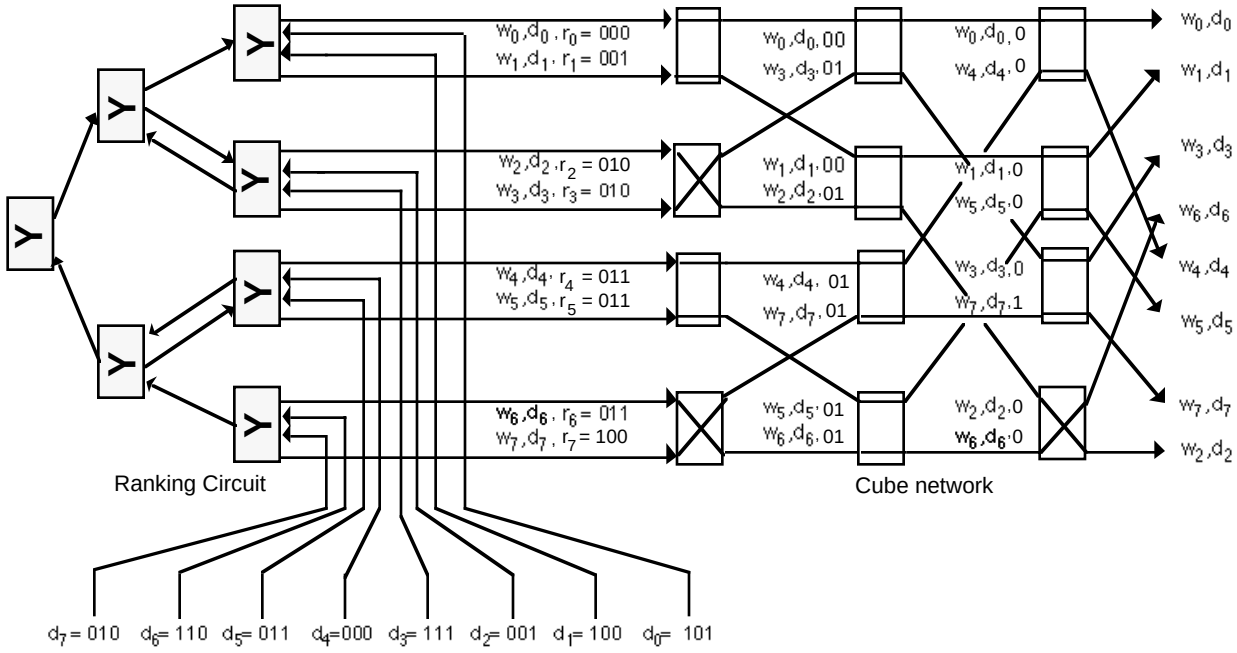

Figure 10. An 8-input 1–sorter.

inputs of S_i (This is a well–known property of a cube network). Now, since R_x and R_y are associated with some two inputs whose destination addresses carry a 1 in their most significant bits, they cannot not be equal, and therefore must differ in at least one of the remaining bit positions $\log n - 1, \ldots, i+1, i$. However, since the ranks are assigned to the inputs of the cube network in increments of 1, we must have $|R_x - R_y| \leq 2^i$, and thus R_x and R_y cannot differ in bit positions $\log n - 1, \log n - 2, \ldots, i+1$. Therefore, they must differ in bit position i, and the statement we set out to prove follows.

The preceding discussion leads to

Theorem 2: An n-input ranking circuit cascaded with an n-input cube network whose switches are set as in Table 1 forms a self–routing 1–sorter. ||

Figure 10 further illustrates how the tree ranking circuit and the cube network are combined to form a 1–sorter. One can also use the ranking circuit of Figure 7 in the construction. The inputs to the cube network are formed by concatenating datum w_i with its destination d_i and rank r_i. The network then uses these ranks along with the destination addresses as outlined in Table 1 to concentrate w_i. It should be noted that the bits of r_i are decoded from right to left as they move from left to right through the network.

V. Performance Analysis

In this section the hardware cost and propagation delay of the self–routing permutation network and Batcher's network will be analyzed. As we assumed before, throughout this section the formulas which are derived below hold only for n which is a power of two, even though they can easily be extended to other values of n.

Self–routing Network

First consider the self–routing network. In obtaining its cost, three types of units will be tallied: (a) 1-bit wide, 2-input crossbar switches, called *switch slices*, (b) 1-bit wide, 2-input adders, called *adder slices*, and (c) simple combinational circuits, called *function slices*. These latter slices are used to determine the state of a switch from the destination information it receives at its two inputs. Not all of these slices are needed in all sections of the self–routing network. The adder slices will be used in the ranking circuit, while the switch and function slices will be used in the cube network, and the center stage of the self–routing network. The cost of each switch or adder slice will be counted as 1, and the cost of a function slice will be assumed to be half the number of its inputs. The propagation delays of all three slices will be taken as 1. These assumptions are based on the fact that a 2-input switch slice can be implemented by two 2×1 multiplexers (which need about four 2-input gates), and that a 2-input adder slice can be implemented by two XOR gates, one AND gate, and one OR gate, roughly equivalent to the gate count of two 2-input multiplexers.

First, consider the number of switch slices in the cube network. Let q denote the width of each data input to a 2-input switch in the network. Referring to Figure 10 it is seen that along with data inputs $w_i; 0 \leq i \leq n/2 - 1$, we should also move the $\log n$-bit destination addresses $d_i; 0 \leq i \leq n/2 - 1$ from the input side to the output side of an $n/2$-input cube network. In addition, each switch in the ith stage carries $\log n/2 - i - 1$ bits of rank information. Thus, we can think of each 2-input switch in the ith stage as consisting of $q + \log n + \log \frac{n}{2} - i - 1$ switch slices. Noting that each stage in an $\frac{n}{2}$-input cube network consists of $\frac{n}{4}$ 2-input switches, and summing the number of switch slices which make up these 2-input switches over all $\log \frac{n}{2}$ stages of the network, we determine the number of switch slices as:

$$\frac{n}{4}\left(\sum_{i=0}^{\log \frac{n}{2}-1}(q + \log n) + \sum_{i=0}^{\log \frac{n}{2}-2}(\log \frac{n}{2} - i - 1)\right) \tag{1}$$

or,

$$\frac{n}{8}\log^2\left(\frac{n}{2}\right) + \left(\frac{q + \log n}{2} - \frac{1}{4}\right)\frac{n}{2}\log\left(\frac{n}{2}\right). \tag{2}$$

Similarly, the number of switch slices in each of the 2-input switches in the center stage of the n-input self–routing network is $q + \log n - 1$; q switch slices for the data, and $\log n - 1$ switch slices for the remaining bits in the destination address (the most significant bit is no longer needed.) Since the center stage consists $n/2$ 2-input switches, the total number of switch slices is

$$\frac{n}{2}(q + \log n - 1). \tag{3}$$

Furthermore, since the ranking circuit does not contain any 2-input switches, we can multiply (2) by 2 and add it to (3) to obtain the total number of switch slices within the two 1–sorters and the center stage. Denoting the total number of switch slices in n-input self–routing permutation network $XB_{\mathrm{SRN}}(n)$, we can thus write

$$\begin{aligned} XB_{\mathrm{SRN}}(n) = 2XB_{\mathrm{SRN}}\left(\frac{n}{2}\right) + 2\left[\frac{n}{2}(\log n - 1)\left(\frac{q + \log n}{2} - \frac{1}{4}\right) + \frac{n}{8}\log^2\left(\frac{n}{2}\right)\right] \\ + \frac{n}{2}(q + \log n - 1) \end{aligned} \tag{4}$$

the solution to which is

$$XB_{\mathrm{SRN}}(n) = \frac{1}{4}n\log^3 n + \frac{q}{4}n\log^2 n + \left(\frac{q}{4} - \frac{1}{4}\right)n\log n \tag{5}$$

switch slices.

As for the adder slices, we need only consider the ranking segment of the 1–sorters in the first stage. From the description of an $n/2$-input ranking circuit, it is easily verified that it consists of $n/2 - 2$ Y–units (not counting the root since it is only wire). Each Y–unit contains two 2-input adders; the maximum sum that the adders can generate for an $n/2$-input ranker is $n/2 - 1$, requiring $\log \frac{n}{2}$ bits for the adder's inputs and outputs. Assuming that the total number of adder slices in an 2-input adder is proportional to the length of its inputs (as in a ripple adder), an $n/2$-input ranking circuit will roughly contain $2(n/2 - 2)\log \frac{n}{2} = (n-4)\log \frac{n}{2}$ adder slices. Thus if $AS_{\mathrm{SRN}}(n)$ denotes the total number of adder slices in the n-input self–routing network, we can write

$$AS_{\mathrm{SRN}}(n) = 2AS_{\mathrm{SRN}}\left(\frac{n}{2}\right) + 2(n-4)(\log n - 1) \tag{6}$$

the solution to which is

$$AS_{\mathrm{SRN}}(n) = n\log^2 n - n\log n + 8\log n - 8n + 8 \tag{7}$$

adder slices.

Finally, we count the number of function slices which are needed to set the switches both in the center stage of the self–routing network, and those in the cube network segment of the 1–sorters. The switches in the center stage of the self–routing network are set by examining a single bit of destination address, and hence the number of function slices for these switches is $n/2$. On the other hand, a 2-input switch in the 1–sorters is set by examining two rank bits and two destination bits as described in Table 1 (this is the worst case), thereby requiring 2 function slices. It follows that the total number of function slices in the two 1–sorters is $2(n/2)\log\frac{n}{2} = n\log\frac{n}{2}$. Denoting the total number of function slices in the entire network by $FS_{\mathrm{SRN}}(n)$, we can now write

$$FS_{\mathrm{SRN}}(n) = 2FS_{\mathrm{SRN}}\left(\frac{n}{2}\right) + 2\left(\frac{n}{2}(\log n - 1)\right) + \frac{n}{2} \tag{8}$$

function slices. The solution to this recurrence is

$$FS_{\mathrm{SRN}}(n) = \frac{n}{2}\log^2 n \tag{9}$$

function slices.

As for the propagation delay $D_{\mathrm{SRN}}(n)$ of the n-input self–routing network, we can write

$$D_{\mathrm{SRN}}(n) = D_{\mathrm{SRN}}(\frac{n}{2}) + D_{\mathrm{ranker}}(\frac{n}{2}) + D_{\mathrm{cube}}(\frac{n}{2}) + D_C. \tag{10}$$

Noting that $D_{\mathrm{ranker}}(\frac{n}{2}) = 2\log^2\frac{n}{2} - 2\log\frac{n}{2}$, $D_{\mathrm{cube}}(\frac{n}{2}) = 2\log\frac{n}{2}$, and $D_C = 1$, it can be shown that

$$D_{\mathrm{SRN}}(n) = 2/3\log^3 n - \log^2 n + 1/3\log n + 1. \tag{11}$$

Batcher's Network

The Batcher's odd-even merge sorter consists $(n\log^2 n)/4 + (n\log n)/4$ compare/swap (CS) units each of which can be viewed as encompassing 2-input switches (swap unit), and a function slice (compare unit). If this sorter is used as a permutation network then, each swap unit must switch some q data bits and $\log n$ destination address bits. Multiplying the total number of bits to be handled with the number of 2-input switches,

$$XB_{\mathrm{Batcher}}(n) = \frac{1}{4}n\log^3 n + \frac{(q+1)}{4}n\log^2 n + \frac{nq}{4}\log n. \tag{12}$$

In order to switch its inputs, each CS unit must compare two $\log n$-bit addresses which can be carried out by using about $\log n$ 1-bit wide comparators (or function slices). Therefore, the total number of function slices in the n-input Batcher's network is

$$FS_{\text{Batcher}}(n) = \frac{n}{4}\log^3 n + \frac{n}{4}\log^2 n. \tag{13}$$

The gate–level delay of the n-input Batcher's network can be determined by multiplying its comparator–level delay [10] by $\log n$:

$$D_{\text{Batcher}}(n) = \frac{1}{2}(\log^3 n + \log^2 n). \tag{14}$$

Table 2 summarizes the results obtained above. The highest order terms for switch, adder, and function slices, the total hardware, and total propagation delay are tabulated for both networks. It is seen that both networks use the same number of switch slices while the self–routing network of this paper uses a smaller order of function slices than Batcher's network. Thus, if as we assumed in the preceding section, the cost of all three slices are identical, then the highest order terms of the total hardware cost of both networks are equal with the constant term multiplying the highest order term in the cost expression of the self–routing network being half of that for Batcher's network. However, function slices are likely to be more complex than the switch slices in an actual implementation. This will make the ratio of the constants multiplying the highest order terms even worse for Batcher's network. As for propagation delay, the table indicates that both networks have the same order of delays with Batcher's network delay being three fourths of the delay of this paper's self–routing network.

Table 2

Cost	self–routing Network	Batcher's Network
Crossbar	$n(\log^3 n)/4$	$n(\log^3 n)/4$
Function + Adder Slices	$3(n\log^2 n)/2$	$n(\log^3 n)/4$
All Hardware	$n(\log^3 n)/4$	$n(\log^3 n)/2$
Propagation Delay	$\frac{2}{3}\log^3 n - \log^2 n + \frac{1}{3}\log n + 1$	$\frac{1}{2}\log^3 n + \frac{1}{2}\log^2 n$

VII. Concluding Remarks

A self–routing permutation network using less hardware than Batcher's sorting networks has been described. The network has a similar structure to the complementary Beneš network, but uses a device, called a 1–sorter, in certain places, rather than crossbar switches, or smaller complementary Benes networks. A detailed design of 1–sorters by using ranking and concentrator devices has been provided. Two different ranking circuits have been described, one by using 2-input adders, and another by using combinational circuits called Y–units. Both ranking circuits have $O(\log^2 n)$ gate delay, the one which uses 2-input adders has $O(n\log n)$ cost, and the other has $O(n)$ cost. However, both results in the same order of cost complexity when used to form a 1–sorter, even though one which uses Y–units is more economical if the constants are taken into account.

The results of this paper show that the use of 1–sorters leads to both reduction in cost and a simpler self–routing scheme for permutation maps when compared to Batcher's sorting networks. Nonetheless, the cost of the self–routing network described here is still higher than the cost of an optimal permutation network by a factor of $O(\log^2 n)$ if the programming hardware cost of the latter is not taken into account. Even though this is an unrealistic assumption, it remains open whether one can obtain a self–routing network whose cost matches that of an optimal permutation network.

REFERENCES

[1] K.E. Batcher, "Sorting networks and their applications," in *Proceedings of 1968 Spring Joint Computer Conference,* pp. 307–314.

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

[3] R. Boppana and C. S. Raghevandra, "On self–routing in Benes and shuffle-exchange networks," in *Proceedings of International Conference on Parallel Processing,*, Vol. 1, Aug. 1988, pp. 196–200.

[4] J.D. Carpinelli and A. Y. Oruç "Parallel set-up algorithms for Clos networks," in *Proceedings of 2nd International Conference on Supercomputing.* May 1987, pp. 321-327, Santa Clara, CA.

[5] C. Clos, "A study of non–blocking switching networks," *The Bell System Technical Journal,* March 1953, pp. 406–424.

[6] T–Y Feng, "A survey of interconnection networks," *IEEE Computer,* Dec. 1981, pp. 12–27.

[7] C.-Y. Chin and K. Hwang, "Packet switching networks for multiprocessor and data flow computers," *IEEE Transaction on Computers,* Vol.'C–33, May 1985, pp. 991–1003.

[8] W.H. Kautz, K.N. Levitt, and A. Waksman, "Cellular interconnection arrays," *IEEE Transactions on Computers,* Vol. C-17, No. 5, May 1968, pp. 443-451.

[9] S.C. Knauer, J.H. O'Neill, and A. Huang, "self–routing switching network," in *Principles of CMOS VLSI Design, A Systems Perspective,* N. Weste and K. Eshraghian, Addison-Wesley, Reading, MA, pp. 424–449.

[10] D.E. Knuth, *The Art of Computer Programming* Reading, MA, Addison–Wesley, pp. 224–225.

[11] D.M. Koppelman, A. Thirumalai, J.F. McDonald, and A. Yavuz Oruç , "The implementation of a triangular permutation network using wafer scale integration," in *Proceedings of International Conference on Computer Design,* October 1986, pp. 269–274.

[12] D.H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transaction on Computers,* Vol.'C–24, Dec. 1975, pp. 1145–1155.

[13] G.F. Lev, N. Pippenger and L. Valiant, "A fast parallel algorithm for routing in permutation networks," *IEEE Transactions on Computers,* Vol. C–30, No. 2, Feb. 1981, pp. 93–100.

[14] M.J. Marcus, "The theory of connecting networks and their complexity: a review," *Proceedings of the IEEE,* Vol. 65, No. 9, September 1977, pp. 1263–1271.

[15] G.M. Masson, G.C. Gingher, and S. Nakamura, "A sampler of circuit switching networks," *IEEE Computer,* June 1979, pp. 32–48.

[16] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network," *JACM,* Vol. 29, No. 3, July 1982, pp. 642–667.

[17] D. Nassimi and S. Sahni, "Parallel algorithms to set up the Beneš permutation network," *IEEE Transactions on Computers,* Vol. C-31, No. 2, Feb. 1982, pp. 148–154.

[18] D. Nassimi and S. Sahni, "A self–routing Beneš network, and parallel permutation algorithms," *IEEE Transactions on Computers,* Vol. C-30, No. 2, May 1981, pp. 157–161.

[19] D.C. Opferman and N.T. Tsao–Wu, "On a class of rearrangeable switching networks," *The Bell System Technical Journal,* May-June 1971, pp. 1579-1600.

[20] A. Yavuz Oruç, "Rearrangeable networks based on double coset decompositions of symmetric groups," in *Proceedings of 21st Annual Conference of Information Sciences and Systems,* John Hopkins University, Baltimore MD, pp. 453–458.

[21] A. Yavuz Oruç and M. Yaman Oruç, "Programming cellular permutation networks through coset decomposition of symmetric groups," *IEEE Transactions on Computers,* July 1987, pp. 802–809.

[22] A. Y. Oruc "Designing cellular permutation networks through coset decompositions of symmetric groups," *Journal of Parallel and distributed Computing*, Academic Press Pub., Vol. 4, No. 2, Aug. 1987, pp. 402-422.

[23] D. S. Parker Jr, "New points of view on three-stage rearrangeable switching networks," in *Proceedings of the Workshop on Interconnection Networks,* Computer Society Pub., 1980, pp. 56–63.

[24] M.C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers,* Vol. C-26, May 1977, pp. 458–473.

[25] H. J. Siegel, "Interconnection networks for SIMD computers," *IEEE Computer,* June 1979, pp. 57-65.

[26] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, D. C. Heath and Company, Lexington, MA.

[27] C. D. Thompson, "Generalized interconnection networks for parallel processor intercommunication," *IEEE Transactions Computers*, Vol. C-27, Dec. 1978, pp. 1119-1125.

[28] A. Waksman, "A permutation network," *Journal of the ACM,* vol. 15, No. 1, January 1968, pp. 159–163.

[29] C. L. Wu, and T. Feng, "On a class of multistage interconnection networks," *IEEE Transactions Computers*, Vol. C-29, Aug. 1980, pp. 694-702.