ADAPTIVE BINARY SORTING SCHEMES AND ASSOCIATED INTERCONNECTION NETWORKS*

Minze V. Chien and A. Yavuz Oruç
Electrical Engineering Department
and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

ABSTRACT

Many routing problems in parallel processing such as concentration and permutation problems can be cast as sorting problems. In this paper we consider the problem of sorting on a new model, called an adaptive sorting network. We show that any sequence of \( n \) bits can be sorted on this model in \( O(\lg^2 n) \) bit-level delay using \( O(n) \) constant fanin gates. This improves the cost complexity of Batcher’s binary sorters by a factor of \( O(\lg^2 n) \) while matching their sorting time. The only other network that can sort binary sequences in \( O(n) \) cost is the network version of columnsort algorithm but this requires excessive pipelining. We also show that any sequence of \( n \) numbers can be sorted on the same model in \( O(\lg^2 n) \) comparator-level delay using \( O(n \lg \lg n) \) comparators. In addition, using binary sorters, we construct permutation networks with \( O(n \lg n) \) bit-level cost, and \( O(\lg^3 n) \) bit-level delay. These results provide the asymptotically least-cost sorting and permutation networks to date. We, of course, note that the well-known AKS sorting network has \( O(\lg n) \) sorting time and \( O(n \lg n) \) cost, but the constants hidden in these complexities are so large that our complexities outperform those of the AKS sorting network until \( n \) becomes extremely large.

Index Terms: adaptive sorting, binary sorter, concentrator, permutation network, sorting network.

*This work is supported in part by the National Science Foundation under Grant No: CCR-8708864, and in part by the Minta Martin Fund of the School of Engineering at the University of Maryland.

All logarithms in this paper are in base 2, and \( \lg n \) denotes \( \log_2 n \).
1 Introduction

Sorting networks have received much interest because of their wide spread use in many computations and algorithms see [Knu73] for an in-depth treatment of sorting networks, and [Ric86,Par89] for recent surveys. These networks are constructed by cascading constant fanin comparator switches, and are termed nonadaptive in that the comparisons at the switches are not predicated on any conditions [Voo74]. Inputs upon entering a sorting network are compared and exchanged at the comparators depending on their relative values as shown in Figure 1. Two parameters that are most commonly used to assess the performance of a sorting network are its cost and depth. The cost of a sorting network is the number of constant fanin comparator switches it contains, and its depth is the maximum number of such switches on a path from an input to an output. The cost and depth of the network in Figure 1 are 5 and 3, respectively.

Many models of sorting exist, but sorting networks seem to have been most resourceful among these in that many parallel sorting algorithms are adaptations of Batcher’s odd-even merge and bitonic sorting networks [Bat68,Knu73], and the more recent AKS sorting network [AKS83,Pat90,Par89]. In particular, the AKS sorting network construction settled a long standing question about the existence of a sorting network with $O(n \lg n)$ cost and $O(\lg n)$ depth and had strong implications on the complexity of sorting on parallel computer models [Lei85]. Despite its theoretical significance, however, the AKS sorting network is far from having any practical value due to the large constants in its cost and depth complexities, and the problem of constructing an $O(n \lg n)$ cost and $O(\lg n)$ depth sorting network with small constants remains a challenging open problem.

In this paper, we examine the possibility of constructing sorting networks whose cost and depth complexities approach these expressions as best as possible. To facilitate this, we consider a new sorting network model, called an adaptive sorting network. The main difference between this model and a nonadaptive sorting
Figure 1: A 4-input sorting network.

network is that the latter is constructed by using only comparators whereas an adaptive sorting network may have other components to check on conditions for comparing and routing its inputs through. Our main problem on this model is to sort an arbitrary binary sequence of \( n \) elements. Apart being significant in and of its own, sorting binary sequences plays quite a central role in concentration, permutation and sorting problems. In fact, the concentration problem is equivalent to sorting binary sequences [Cor86], and the permutation and sorting problems can be broken into a sequence of sorting steps on binary sequences.

Obviously, the well-known zero-one principle dictates that any nonadaptive network of comparators that sorts an arbitrary binary sequence also sorts any “totally ordered” set of elements [Knuth73]. As such, an \( O(n \lg n) \) cost and \( O(\lg n) \) depth nonadaptive binary sorting network with small constants in the order expressions, i.e., one that sorts all binary sequences of length \( n \), will have strong implications on permuting and sorting in general, but this seems highly unlikely given the countless unsuccessful attempts on the problem.

As an alternative, we explore in this paper the possibility of constructing adaptive networks to sort binary sequences within optimal bounds. More specifically, we have the following results:
1) An adaptive binary sorting network with $O(n \lg n)$ bit-level cost and $O(\lg^2 n)$ bit-level depth using a prefix adder scheme,\(^1\)

2) An adaptive binary sorting network with $O(n \lg n)$ bit-level cost and $O(\lg^2 n)$ bit-level depth using a multiplexed merging scheme,

3) An adaptive binary sorting network with $O(n \lg \lg n)$ cost, $O(\lg^2 n)$ depth, and $O(\lg^3 n)$ sorting time without pipelining or $O(\lg^2 n)$ sorting time with pipelining, all in bit level, using a time-multiplexed sorting scheme,

4) An adaptive binary sorting network with $O(n)$ cost, $O(\lg^2 n)$ depth, and $O(\lg^3 n)$ sorting time without pipelining or $O(\lg^2 n)$ sorting time with pipelining, all in bit level, using a simplified time-multiplexed sorting scheme.

These network constructions are all at least as good or superior to the binary versions of the best-known sorting networks. The first two have $O(n \lg n)$ bit-level cost that is only matched by the AKS binary sorting network but with an impractically large constant. The third has $O(n \lg \lg n)$ bit-level cost which is less than the cost of the AKS binary sorting network even if the constants are ignored. The last one has $O(n)$ bit-level cost which is as good as the network version of columnsort [Lei85], in which the sorting steps are implemented by $n/\lg^2 n$-input Batcher’s sorters and inputs are pipelined. However, we must note that while the columnsort network requires inputs to each of its sorters being separately pipelined, our last network construction requires that the inputs be pipelined through a single $n/\lg n$-input sorter. Their bit-level depth and sorting time complexities match those of Batcher’s binary sorting networks and binary columnsort networks.

The rest of this paper is organized as follows. Section 2 describes our adaptive sorting network models. Section 3 presents our binary sorting networks. Section 4 describes an adaptive network for sorting arbitrary numbers, and other totally ordered sets of elements. Section 5 describes how binary sorters can be used as

\(^1\)Here bit-level cost refers to the number of constant fanin logic gates, and bit-level depth is the maximum number of such gates on a path from an input to an output.
concentrators and compare them with other concentrators. Section 6 constructs a permutation network using adaptive binary sorters, and the paper is concluded in Section 7.

2 Our Network Models

In this section, we describe the two models we adapted in this paper along with their building blocks. First, we sketch the building blocks needed in our models.

A. 2-way swapper

A 2-way swapper network, as shown in Figure 2(a), swaps two halves of inputs to outputs, when its control signal is enabled. An \( n \)-input 2-way swapper network consists of a shuffle connection, a single stage of \( n/2 \) \( 2 \times 2 \) switches, a reversed shuffle connection, and a single control signal.

B. 4-way swapper
A 4-way swapper network, as shown in Figure 2(b), swaps four quarters of inputs to their outputs, when its select signals are set accordingly. An \( n \)-input 4-way swapper network consists of a shuffle connection, a single stage of \( n/4 \) \( 4 \times 4 \) switches, a reversed shuffle connection, and two select signals.

C. Multiplexers

An \((m, 1)\)-multiplexer is a device which can select and connect one of its \( m \) inputs to its only output. It can be constructed with a balanced binary tree with \( \lg m \) levels of 2-input OR gates and \( m \) leaf nodes of 2-input AND gates each taking one select control signal. An \((n, k)\)-multiplexer can select and connect one of \( n/k \) groups of inputs, where each group consists of \( k \) inputs, to a \( k \)-output channel according to a \( \lg n/k \)-bit select control signals, where \( k \leq n \). An \((n, k)\)-multiplexer can be formed by coupling \( k \) \((n/k, 1)\)-multiplexers. The select control signals can be generated by a \( \lg n/k \)-bit decoder. A \((16, 4)\)-multiplexer, without selecting signals and the 4-bit decoder, is shown in Figure 3(a).

D. Demultiplexers

A \((1, 2)\)-demultiplexer is a device which can connect its only input to either one of its two outputs. A \((1, m)\)-demultiplexer is a device which can connect its only input to one of its \( m \) outputs. A \((1, m)\)-demultiplexer can be constructed by a balanced binary tree with \( \lg m \) levels of \((1, 2)\)-multiplexers, each taking a select control signal. A \((k, n)\)-demultiplexer connects its \( k \) inputs to one of its \( n/k \) groups of outputs according to its \( \lg n/k \)-bit select control signals, where \( k \leq n \). An \((n, k)\)-demultiplexer can be formed by coupling \( k \) \((1, n/k)\)-demultiplexers. The select control signals can be generated by a \( \lg n/k \)-bit decoder. A \((4, 16)\)-demultiplexer, without selecting signals and its 4-bit decoder, is shown in Figure 3(b).

As stated in the introduction, an adaptive sorting network allows not only comparators but also other components in its construction. More precisely, we define two adaptive sorting network models as follows:

*Adaptive Sorting Network Model A*
In this model, inputs are not pipelined and besides comparator switches, any combinational circuits, like adders, 2-way swappers or 4-way swappers described before, are permitted as building components. Thus, the resulting sorting networks of this model are combinational circuits.

Adaptive Sorting Network Model B

In this model, inputs are allowed to be pipelined. Besides components allowed in Model A, both \((n,k)\)-multiplexers and \((k,n)\)-demultiplexers are also permitted in the construction.

3 Adaptive Binary Sorting Networks

We present four adaptive binary sorting networks, the first two belong to the adaptive sorting network model A while the last two belong to adaptive sorting network model B. For ease of discussion, but with no loss of generality, we shall
assume that all our sorting networks use a power of two inputs.

A. Network 1 (Prefix Binary Sorter)

The first binary sorting network is based on a variant of odd-even merge sorting network [Bat68]. The construction of this sorting network for 16 inputs is shown in Figure 4(b), and can easily be extended to larger numbers of inputs. This sorter construction is similar to Batcher’s odd-even merge sorting network (Figure 4(a)), except that the two $n/2$-input sorters in Batcher’s network are replaced by $n/2$ 2-input sorters, the even and odd $n/2$-input mergers are replaced by $n/2$-way mergers, and the merging of sorted subsequences is done by a network, called a balanced merging block [DPRS83,Rud85,DPRS89]. In other words, the outputs of the binary comparators in the first stage contain $n/2$ sorted sequences, each with two elements, and the rest of the network merges them by using an odd-even merge scheme. The $n/2$-way mergers are actually $n/2$-input sorters, where each of the $n/2$ sorted sequences to be merged contains a single element. (In fact a moment’s reflection reveals that merging sets, each containing only one element amounts to sorting them.) As stated in Figure 4(b), the first stage of comparators and the shuffle connection are redundant and are only included to demonstrate the extension of Batcher’s odd-even merge sorting networks.

Obviously, the balanced merging block that follows the two $n/2$-way mergers in the network of Figure 4(b) is more complex than $n/2-1$ 2-input comparators used in Batcher’s odd-even merge sorter. The tradeoff is that the sorting problem on the input side in this case is much simpler; we only need $n/2$ 2-input sorters (i.e., 2-input comparators) rather than two $n/2$-input sorters.

For binary sequences, the balanced merging block sends down the 1’s to the lower half where they belong and similarly 0’s are sent up where they belong. The balanced merging block used on the right side of Figure 4(b) has $O(n \log n)$ bit-level cost and $O(\log n)$ bit-level depth. If the two $n/2$-way merging networks are recursively replaced by half-size odd-even merge sorting networks, then a sorting
Figure 4: 16-input odd-even merge sorting networks.
network with $O(n \log^2 n)$ bit-level cost and $O(\log^2 n)$ bit-level depth is obtained.

This odd-even merge sorting scheme works for arbitrary numbers. For binary inputs (one bit inputs), the cost of the balanced merging block can be reduced from $O(n \log n)$ to $O(n)$ by observing the following facts.

**Definition 1:** Let $A_n$ be the set of all binary sequences of length $n$ starting with any multiple of 00 or any multiple of 11, followed by any multiple of 01 or, any multiple of 10, and that followed by any multiple of 00 or, any multiple of 11. This is stated more precisely by the regular expression

$$A_n = \{0, 1\}^n \cap [(00)^* + (11)^*)((01)^* + (10)^*)((00)^* + (11)^*)].$$

As an example, 0000/1010, 00/1010/11, 101010/11, 00/0101/11, 11111111 are all elements of $A_8$.

**Remark:** Note that zero multiples of 00, 01, 10, and 11 are allowed. Note also that any sorted binary sequence of length $n$ belongs to $A_n$.

**Theorem 1:** Let $X_U$ and $X_L$ be any two ascendingly sorted binary sequences of length $n/2$. If $X_U$ and $X_L$ are concatenated and shuffled then the resulting sequence, $X_n$, belongs to $A_n$.

**Proof:** Let $n_1, n_2$ denote the numbers of 0’s and 1’s in $X_U$, respectively, and $m_1, m_2$ denote the numbers of 0’s and 1’s in $X_L$, respectively, where $0 \leq n_1, n_2, m_1, m_2 \leq n/2$. If $n_1 \leq m_1$ then $X_n$ must start with $n_1$ multiples of 00, followed by $(m_1 - n_1)$ multiples of 10, and that followed by $m_2$ multiples of 11, which is an element of $A_n$. If $n_1 > m_1$ then $X_n$ must start with $m_1$ multiples of 00, followed by $(n_1 - m_1)$ multiples of 01, and that followed by $n_2$ multiples of 11, which is also an element of $A_n$.

**Definition 2:** A binary sequence is called clean-sorted if its elements are all identical, i.e., either all 0’s or all 1’s.

**Theorem 2:** Let $Y_U$, and $Y_L$ denote the upper and lower halves of outputs after the first stage of $n/2$ comparators in the balanced merging block of the network.
in Figure 4(b). For any binary sequence $Z$ in $A_n$ that enters the inputs of the balanced merging block, one of $Y_U$ and $Y_L$ must be clean-sorted, and the other must contain a binary sequence that belongs to $A_{n/2}$.

**Proof:** Consider an arbitrary binary sequence $Z$ in $A_n$. Let $Z_a, Z_b$ and $Z_c$ denote the three parts of $Z$, respectively, as given in Definition 1, and $k_a, k_b$ and $k_c$ denote the numbers of elements in $Z_a, Z_b$ and $Z_c$, in that order. We have $0 \leq k_a, k_b, k_c \leq n$, and consider two cases.

Case (1): If $k_b = 0$, then, obviously, one of $Y_U$ and $Y_L$ must be clean-sorted.

Case (2): If $k_b \neq 0$, then we consider two cases:

Case (2.1): If $k_a \geq \frac{n}{2}$ and $k_c < \frac{n}{2}$, then $Y_U$ must be all 0’s and $Y_L$ must contain a binary sequence that belongs to $A_{n/2}$. Likewise, if $k_a < \frac{n}{2}$ and $k_c \geq \frac{n}{2}$, then $Y_L$ must be all 1’s and $Y_U$ must contain a binary sequence that belongs to $A_{n/2}$.

Case (2.2): If both $k_a < \frac{n}{2}$ and $k_c < \frac{n}{2}$, then the elements of $Z_b$ cannot all be contained in one half of the inputs of the merging network. Let $Z_{bu}$ and $Z_{bl}$ be the two parts of $Z_b$ in the upper and lower halves of inputs. Let $k_{bu}$ and $k_{bl}$ denote the numbers of elements in $Z_{bu}$ and $Z_{bl}$, respectively. We consider three cases:

Case (2.2.1): If $k_{bl} = k_{bu}$ then after the first stage of comparators, the 1’s are interchanged with 0’s, and $Y_U$ must be all 0’s while $Y_L$ must be all 1’s.

Case (2.2.2): If $k_{bl} > k_{bu}$ then, after the first stage of comparators, $Y_U$ must be all 0’s, and $Y_L$ must contain a binary sequence that belongs to $A_{n/2}$.

Case (2.2.3): If $k_{bl} < k_{bu}$ then, after the first stage of comparators, $Y_L$ must be all 1’s, and $Y_U$ must contain a binary sequence that belongs to $A_{n/2}$.

Given this result, the balanced merging block can be simplified by devising a circuit that detects which half of outputs is clean-sorted after the first stage of $n/2$ comparators in the balanced merging block, and then sort only that half which is not sorted. Since, by Theorem 2, the unsorted half belongs to $A_{n/2}$, we can construct this circuit, which we call a *patch-up network*, recursively as shown in Figure 5 for $n = 16$. The detection of the unsorted half of outputs is carried out by a simple $\lg n$-bit prefix adder which gives the count of the number of 1’s in the
Figure 5: A 16-input adaptive binary sorting network using prefix adder scheme. The entire input sequence. It does this by recursively adding the numbers of 1’s in the two half-size input sequences as indicated in the figure.

Also indicated in the figure, the patch-up network is recursively constructed from a half-size patch-up network, one stage of comparators in a balanced merging block, and some multiplexing and demultiplexing circuitry. The comparators move the 0’s in the lower half up and 1’s in the upper half down to where they belong whenever the compared bits are different. As implied by Theorem 2, after the comparator stage, at least one half of outputs will be clean-sorted, and the other half must be a sequence that belongs to $A_{n/2}$. By examining the most significant bit of the result from the prefix adder, we can determine if the number of 1’s is greater than or equal to $n/2$. If so, then the lower half of outputs must be clean-sorted and we have the upper half left unsorted. Otherwise, the lower half of outputs must be unsorted and the upper half must be clean-sorted.

The selection of the unsorted half of outputs is carried out by an $n$-input, 2-way swapper. The swapper network uses the most significant bit of the sum from the prefix adder as a select input to channel the unsorted half of outputs to the next
level of the patch-up network which, by induction, is assumed to sort any binary sequence that belongs to $\mathcal{A}_{n/2}$. If its select input is 0, then the inputs are connected to the outputs straight across. If it is 1, then the upper half of inputs is connected to the lower half of outputs, and the lower half of inputs is connected to the upper half of outputs.

Once the unsorted half is selected and sorted by the patch-up network then if the select input of the 2-way swapper in the last stage is 1 then the outputs of the patch-up network are switched to the upper half of network’s outputs, and otherwise are connected to its lower half of outputs. Given that the half-size patch-up network sorts its inputs onto its outputs, it is easy to verify that the swapper networks operating as described will produce a sorted sequence.

**Corollary:** The network in Figure 5 (its $n$-input version) sorts any binary sequence of $n$ elements in ascending order.

**Proof:** The proof immediately follows from Theorems 1 and 2.

Since the unsorted half of outputs after the comparator stage contains a sequence that belongs to $\mathcal{A}_{n/2}$, we can repeat this multiplex and sort process recursively. The next most significant bit in the sum computed by the prefix adder is used in the next level of the patch-up network for swapper select, and so on. For example, for a 256-input patch-up network, suppose that the number of 1’s in the sequence is 127 (01111111 in binary). The most significant bit being 0 implies that there are less than $n/2 = 128$ of 1’s and the upper half of outputs must be all 0’s. Hence, we connect the lower half of outputs into a 128-input patch-up network. Given that the second most significant bit is 1, the unsorted part of the sequence contains more 1’s than 0’s, and its lower half must be clean-sorted after passing through the comparator stage in the 128-input patch-up network, while its upper half must be connected to a 64-input patch-up network. This “compare and swap” process is recursively repeated over until a size 2 patch-up network is encountered at which time it is replaced by a 2-input comparator.
Let $C(n)$ and $D(n)$ denote the bit-level cost and depth of the network, respectively. Directly from Figure 5 we have

$$
C(n) = 2C(n/2) + C_a(\lg n) + C_p(n) \tag{1}
$$

$$
D(n) = D(n/2) + D_a(\lg n) + D_p(n) \tag{2}
$$

where $C_a(\lg n)$ and $D_a(\lg n)$ denote the cost and depth of a $\lg n$-bit prefix adder and $C_p(n)$ and $D_p(n)$ denote the cost and depth of the patch-up network. The cost and depth of a $\lg n$-bit prefix adder are $3\lg n$ and $2\lg \lg n$, respectively [CLR90]. The cost and depth of the patch-up network can be computed from its recursive construction as

$$
C_p(n) = 3n/2 + C_p(n/2) = O(n) \tag{3}
$$

$$
D_p(n) = 3 + D_p(n/2) = O(\lg n), \tag{4}
$$

where the $3n/2$ and 3 terms account for the bit-level cost and depth of the comparators, and 2-way swapper networks, respectively. The solutions of these recurrences give $C_p(n) \leq 3n$, $D_p(n) \leq 3\lg n$.

Substituting the cost and depth expressions for the prefix adder and patch-up network into Equations 1 and 2, and solving the recurrences shows that the network in Figure 5 has $3n \lg n + O(\lg^2 n)$ bit-level cost and $3\lg^2 n + 2\lg n \lg \lg n$ bit-level depth, respectively.

**B. Network 2 (Mux-Merger Binary Sorter)**

While the prefix binary sorter has $O(n \lg n)$ bit-level cost and $O(\lg^2 n)$ bit-level depth, its design relies on two kinds of elements, a prefix adder and a multiplexer/demultiplexer circuit. Our second binary sorting network eliminates the need for the prefix adder and is based on the following observation.

**Definition 3:** A binary sequence is called bi-sorted if each of its two halves is sorted. ||
**Theorem 3:** If a bi-sorted binary sequence is cut into four equal-size subsequences, then at least two of the subsequences will be clean-sorted and the other two, when concatenated will form a bi-sorted sequence.

**Proof:** Let $X_n$ denote any bi-sorted binary sequence of size $n$. Let $X_{q1}$, $X_{q2}$, $X_{q3}$ and $X_{q4}$ denote the four quarters of $X_n$ from top to bottom, respectively, and $X_U$ and $X_L$ denote the upper and lower halves of $X_n$ (Refer to Figure 6). The proof of the statement requires checking out four cases which are identified with the binary values of the uppermost elements of $X_{q2}$ and $X_{q4}$. If there are more 0’s than 1’s in $X_U$, then the uppermost element of $X_{q2}$ must be 0, $X_{q1}$ must contain all 0’s and $X_{q2}$ must be a sorted sequence of size $n/4$. On the other hand, if the uppermost element of $X_{q2}$ is 1 then there are more 1’s in $X_U$, and hence $X_{q2}$ must contain all 1’s, and $X_{q1}$ must be a sorted sequence of size $n/4$. Similar statements hold for $X_{q3}$ and $X_{q4}$. Therefore, at least two of $X_{q1}$, $X_{q2}$, $X_{q3}$ and $X_{q4}$ must be clean-sorted and the other two must form a bi-sorted sequence when concatenated. ||

Thus, if a binary sequence of $n$ inputs is bi-sorted using two $n/2$-input binary sorters then, the proof of Theorem 3 suggests how to identify the two quarters...
that form a bi-sorted sequence of size $n/2$ when concatenated, as well as the all zero or all one quarters. The network in Figure 6 is constructed based on this fact, where the network enclosed by the rectangle in dash lines, and called Mux-merger, merges the bi-sorted sequence at the outputs of the two sorters into a sorted sequence.

Table 1 lists the possible patterns of bi-sorted sequences and shows how the Mux-Merger selects the quarter-size subsequences for each pattern. The variable entries in the table denote various points in the network as marked in Figure 6. As described in the proof of Theorem 3, the middle two bits of the two sorted halves of outputs can be one of four binary patterns, 00, 01, 10, 11. For each of these, the patterns of $X_{q1}$, $X_{q2}$, $X_{q3}$ and $X_{q4}$ are uniquely determined, and the selections of the quarters can be made accordingly as shown in the table. Furthermore, since the concatenated two quarters form a bi-sorted sequence the same Mux-Merge process can be applied recursively. The selections described in the table can be realized by using $n$-input, 4-way swapper circuits, called IN-SWAP and OUT-SWAP, with their select control described as in the table.

As an example, Figure 7 shows the operations of a 16-input Mux-merger, for a 16-element bi-sorted binary sequence, 00111111/00000111. Since the two middle elements are 1 and 0, by Table 1, IN-SWAP circuit moves $X_{q1}$ to $Y_{q3}$, $X_{q2}$ to $Y_{q2}$, $X_{q3}$ to $Y_{q1}$, and $X_{q4}$ to $Y_{q4}$. The resulting 8-element bi-sorted sequence $Y_{q1}\{Y_{q4}$, i.e., 00110111, then becomes the input for the next level of Mux-merger. The process is repeated through the next level of Mux-Merger, and finally, the multiplexer, OUT-SWAP reorders the four sorted quarters to form a sorted output.

The entire sorting network can be constructed using IN-SWAP and OUT-SWAP circuits if the half-size sorters in Figure 6 are recursively replaced by the same sorter construction. An example is depicted in Figure 8 for $n = 16$.

---

2The symbol * used in the table denotes a concatenation operator.
Table 1
Behavior of Mux-Merger

<table>
<thead>
<tr>
<th>Select inputs</th>
<th>Input pattern</th>
<th>IN-SWAP select</th>
<th>OUT-SWAP select</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>$X_{q1}$ and $X_{q3}$ are all 0's, $X_{q2} \times X_{q4}$ is bi-sorted</td>
<td>$X_{q1}$ to $Y_{q1}$, $X_{q2}$ to $Y_{q3}$, $X_{q3}$ to $Y_{q2}$, and $X_{q4}$ to $Y_{q4}$</td>
<td>straight connection</td>
</tr>
<tr>
<td>01</td>
<td>$X_{q1}$ is all 0's, $X_{q4}$ is all 1's and $X_{q2} \times X_{q3}$ is bi-sorted</td>
<td>$X_{q1}$ to $Y_{q1}$, $X_{q2}$ to $Y_{q3}$, $X_{q3}$ to $Y_{q4}$, and $X_{q4}$ to $Y_{q2}$</td>
<td>$Z_{q1}$ to $W_{q1}$, $Z_{q2}$ to $W_{q4}$, $Z_{q3}$ to $W_{q2}$, and $Z_{q4}$ to $W_{q3}$</td>
</tr>
<tr>
<td>10</td>
<td>$X_{q1} \times X_{q4}$ is bi-sorted, $X_{q2}$ is all 1's, and $X_{q3}$ is all 0's</td>
<td>$X_{q1}$ to $Y_{q3}$, $X_{q2}$ to $Y_{q2}$, $X_{q3}$ to $Y_{q1}$, and $X_{q4}$ to $Y_{q4}$</td>
<td>$Z_{q1}$ to $W_{q3}$, $Z_{q2}$ to $W_{q4}$, $Z_{q3}$ to $W_{q2}$, and $Z_{q4}$ to $W_{q3}$</td>
</tr>
<tr>
<td>11</td>
<td>$X_{q1} \times X_{q3}$ is bi-sorted, $X_{q2}$ and $X_{q4}$ are all 1's</td>
<td>$X_{q1}$ to $Y_{q3}$, $X_{q2}$ to $Y_{q2}$, $X_{q3}$ to $Y_{q4}$, and $X_{q4}$ to $Y_{q1}$</td>
<td>$Z_{q1}$ to $W_{q3}$, $Z_{q2}$ to $W_{q4}$, $Z_{q3}$ to $W_{q1}$, and $Z_{q4}$ to $W_{q2}$</td>
</tr>
</tbody>
</table>

Figure 7: An example showing the operations of a 16-input Mux-merger.
The cost and depth complexities of this network construction can be expressed as

\[ C(n) = 2C(n/2) + C_m(n) \]  (5)
\[ D(n) = D(n/2) + D_m(n) \]  (6)

where \( C(n) \) and \( D(n) \) denote the cost and depth of the entire network respectively, and \( C_m(n) \) and \( D_m(n) \) denote the cost and depth of the Mux-merger. Assuming that a 4-way multiplexer exacts unit cost and unit delay, it is easy to check that the bit-level cost and depth of the Mux-merger are \( 4n \) and \( 2\lg n \), respectively. Thus, the bit-level cost and depth of the overall network are \( 4n\lg n \) and \( 2\lg^2 n \), respectively.

C. Network 3 (Fish Binary Sorter)

The binary sorters described so far both have \( O(n\lg n) \) cost. To further reduce the cost, the network shown in Figure 9 (called the Fish binary sorter due its resemblance to fish) can be used. In this network, we use a multiplexing scheme proposed in [LO92] to obtain compact permutation networks. The idea is to mul-
Figure 9: A 16-input adaptive binary sorting network using a time multiplexing scheme, where $k = 4$.

To multiplex the binary inputs, in time, through a binary sorting network with a smaller number of inputs. Any binary sorting network including those described in the earlier subsections can be used in this kind of multiplexed sorting. The input sequence is first arbitrarily divided into $k$ groups of $n/k$ elements. Each group of inputs is run through an $(n, n/k)$-multiplexer ($(n, n/k)$-MUX) sequentially, and sorted by an $n/k$-input binary sorter. The sorted $n/k$-element sequences are then moved through an $(n/k, n)$-demultiplexer ($(n/k, n)$-DEMUX) circuit to the inputs of an $n$-input $k$-way merging network, where the first $n/k$ inputs of the merging network are occupied by the first sorted $n/k$-element sequence, the second $n/k$ inputs are occupied by the second sorted $n/k$-element sequence, and so on.

The $n$-input $k$-way merging network can be, straight forwardly, implemented in terms of Mux-mergers by using an iterative construction. The first stage of the merger uses $k/2$ 2-way, $2n/k$-input Mux-mergers to merge pairs of $n/k$-element sorted sequences, the second stage uses $k/4$ 2-way, $4n/k$-input Mux-mergers to merge pairs of $2n/k$-element sorted sequences, and in general, the $i$th stage uses $k/2^i$ 2-way, $2^i n/k$-input Mux-mergers to merge pairs of $2^{i-1} n/k$-element sorted sequences, $1 \leq i \leq \lg k$. For the network given in Figure 9, $n = 16$, $k = 4$, $1 \leq i \leq 2$. 
Let $C(n, k)$ and $D(n, k)$ denote the cost and depth of this network, respectively. It follows from the preceding discussion that

$$C(n, k) = C_d(n, n/k) + C_s(n/k) + C_w(n, k)$$  \hspace{1cm} (7)
$$D(n, k) = D_d(n, n/k) + D_s(n/k) + D_w(n, k),$$  \hspace{1cm} (8)

where $C_d(n, n/k)$ and $D_d(n, n/k)$ denote the cost and depth of the $(n, n/k)$-multiplexer and $(n/k, n)$-demultiplexer circuits combined together, $C_s(n/k)$ and $D_s(n/k)$ denote the cost and depth of the $n/k$-input binary sorter, and $C_w(n, k)$ and $D_w(n, k)$ denote the cost and depth of the $k$-way merger. Now, both the $(n, n/k)$-multiplexer and $(n/k, n)$-demultiplexer as described in previous section have $n$ cost and $\log k$ depth. If we use the last sorting network construction of the preceding section then the cost and depth of the $n/k$-input sorter are $4n/k \log n/k$ and $2 \log^2 n/k$, respectively. As for the $k$-way merger, summing up the costs and depths of the Mux-mergers in its construction,

$$C_w(n, k) = \sum_{i=1}^{\log k} \frac{k \cdot 2^{i+1} n}{2^i k} = 2n \log k$$  \hspace{1cm} (9)
$$D_w(n, k) = \sum_{i=1}^{\log k} 2 \log n \cdot \frac{2^i n}{k} \leq 2 \log n \log k$$  \hspace{1cm} (10)

Substituting these into Equations 7 and 8 we find

$$C(n, k) = 2n + \frac{4n}{k} \log \frac{n}{k} + 2n \log k,$$  \hspace{1cm} (11)
$$D(n, k) = 2 \log k + 2 \log^2 \frac{n}{k} + 2 \log n \log k,$$  \hspace{1cm} (12)

and minimizing these expressions with respect to $k$, we obtain

$$C(n, \log n) \leq 6n + 2n \log \log n = O(n \log \log n),$$  \hspace{1cm} (13)
$$D(n, \log n) \leq 2 \log \log n + 2 \log^2 n + 2 \log n \log \log n$$
$$= O(\log^2 n)$$  \hspace{1cm} (15)
We note that the inputs to be sorted are multiplexed in this binary sorter design, and hence the depth of the network does not give the time it takes to complete the sorting. But given that the depths of all the components in this network are known, its sorting time, $T(n, k)$, can be determined by noting that the $k$ groups of $n/k$ bits must be multiplexed through the $n/k$-input sorter. Thus,

\[
T(n, k) = k(D_d(n, n/k) + D_s(n/k)) + D_w(n, k) \\
\leq 2k \log k + 2k \log^2 \frac{n}{k} + 2 \log n \log k
\]

and letting $k = \log n$ gives

\[
T(n, \log n) \leq 2 \log^3 n + 4 \log n \log \log n - 2 \log n \log^2 n \\
= O(\log^3 n).
\]

The sorting time can be reduced to $O(\log^2 n)$ by noting that the $k$ groups of $n/k$ inputs can be pipelined through the $n/k$-input sorting network. In this case, the sorting network is viewed as a $\log^2 n/k$ segment pipeline, where each segment is a constant fanin, unit delay circuit. Therefore, the sorting time $T_{pip}(n, k)$ is given by

\[
T_{pip}(n, k) = O(\log^2 \frac{n}{k}) + O(k) + O(\log k) + O(\log n \log k)
\]

where the $O(\log^2(n/k))$ and $O(\log k)$ terms account for the time for the first group of $n/k$ elements to exit the pipeline, and the $O(k)$ term accounts for the time that the remaining $k - 1$ groups of $n/k$ elements need to get through the pipeline. The $O(\log n \log k)$ term is the merging time. Letting $k = \log n$ gives $T_{pip}(n, \log n) = O(\log^2 n)$.

**D. Network 4 (Simplified Fish Binary Sorter)**

It can be seen from Equation 13 that the $O(n \log \log n)$ cost complexity of the fish binary sorter construction is due to the cost of the $n$-input $k$-way merging network. To further reduce the cost, an extension of Mux-merging scheme can be used for the $n$-input $k$-way merging network, as shown in Figure 10. For this, we first generalize the notion of a bi-sorted binary sequence.
**Definition 4:** A binary sequence is called $k$-sorted if it consists of $k$ equal-sized sorted binary subsequences. 

**Definition 5:** A $k$-sorted binary sequence is called clean $k$-sorted if each of its $k$ equal-sized sorted binary subsequences is clean-sorted, i.e., each contains only 0’s or only 1’s.

**Theorem 4:** If, in a $k$-sorted binary sequence, each of the $k$ equal-sized sorted binary subsequences is cut into two half-size subsequences, then at least $k$ of the half-sized subsequences will be clean-sorted, which if concatenated will form a clean $k$-sorted sequence, and the remaining $k$ half-sized subsequences, when concatenated will form a $k$-sorted sequence.

**Proof:** The proof is a direct extension of the proof of Theorem 3.

Using Theorem 4, we can then extend the 2-way Mux-merger to a $k$-way Mux-merger. In an $n$-input $k$-way Mux-merger, a $k$-sorted input sequence of size $n$, which consists of $k$ sorted subsequences of size $n/k$, is first separated into two halves with $k n/k$-input 2-way swappers, called $k$-SWAP. By connecting the middle bit of each sorted subsequence as the select signal, each of these $k n/k$-input 2-way swappers sends the clean-sorted halves up and the other halves down. As a result, the upper $n/2$ outputs consist of $k$ clean-sorted subsequences of size $n/2k$ and thus, forming a clean $k$-sorted sequence; and, the lower $n/2$ outputs consist of $k$ sorted subsequences of size $n/2k$ and thus, forming a $k$-sorted sequence of size $n/2$.

Since the upper $n/2$-size sequence is clean $k$-sorted which consists of $k$ clean-sorted subsequences of size $n/2k$, by sorting each leading bit of these subsequences and then sending each subsequence to its corresponding sorted position, a sorted output can be obtained by employing a $k$-input sorter, an $(n/2, n/2k)$-multiplexer and an $(n/2k, n/2)$-demultiplexer. Assuming the lower $n/2$ outputs of the $k$-SWAP can be merged recursively and thus sorted, we can use an $n$-input 2-way Mux-merger at the last stage to merge these two sorted sequences and obtain a sorted output. As an example, Figure 11 illustrates the operation of an $n$-input $k$-way Mux-merger.
Figure 10: An n-input adaptive binary sorting network using a time multiplexing scheme and a k-way Mux-merger.

for \( n = 16 \) and \( k = 4 \).

The cost \( C(n, k) \) and depth \( D(n, k) \) of this simplified network can be expressed as

\[
C(n, k) = C_d(n, n/k) + C_s(n/k) + C_{km}(n, k),
\]
\[
D(n, k) = D_d(n, n/k) + D_s(n/k) + D_{km}(n, k),
\]

where, as in previous subsection, \( C_d(n, n/k) \) and \( D_d(n, n/k) \) denote the cost and depth of the \((n, n/k)\)-multiplexer and \((n/k, n)\)-demultiplexer circuits combined together, \( C_s(n/k) \) and \( D_s(n/k) \) denote the cost and depth of the \(n/k\)-input binary sorter, and \( C_{km}(n, k) \) and \( D_{km}(n, k) \) denote the cost and depth of the \(n\)-input \(k\)-way Mux-merger.

It follows from Figure 10 that

\[
C_{km}(n, k) = C_{SWAP}(n) + C_{cs}(n/2, k) + C_{km}(n/2, k) + C_m(n),
\]
\[
D_{km}(n, k) = D_{SWAP}(n) + \max(D_{cs}(n/2, k), D_{km}(n/2, k)) + D_m(n),
\]
where $C_{SWAP}(n)$ and $D_{SWAP}(n)$ denote the cost and depth of the $n$-input $k$-SWAP circuit, $C_{cs}(n/2,k)$ and $D_{cs}(n/2,k)$ denote the cost and depth of the $n/2$-input $k$-way clean-sorter, and $C_m(n)$ and $D_m(n)$ denote the cost and depth of the $n$-input $2$-way Mux-merger, respectively.

Since the $k$-SWAP circuit can be constructed with $k$ $n/k$-input $2$-way swappers, its cost $C_{SWAP}(n)$ is $n/2$ and its depth $D_{SWAP}(n)$ is $1$. The $n/2$-input $k$-way clean-sorter can be constructed with a $k$-input binary sorter, sorting the leading bits of each of the $k$ clean-sorted subsequences, and followed with a dispatching circuit to send each subsequence to its sorted position. This dispatching circuit can be constructed by cascading an $(n/2,n/2k)$-multiplexer and an $(n/2k,n/2)$-demultiplexer. If our Network 2 is used as the $k$-input binary sorter, its cost and depth will be $4k \lg k$ and $2 \lg^2 k$, respectively. The cost and depth of the dispatching circuit can thus be estimated as $n$ and $2 \lg n$, respectively. The cost and depth of the $n$-input $2$-way Mux-merger are $4n$ and $2 \lg n$, respectively.
Substituting all these into Equations 22 and 23 we find

\[ C_{km}(n, k) = \frac{n}{2} + 4k \log k + n + C_{km}(n/2, k) + 4n \]  
\[ \leq 11n + 8k \log^2 k \]  
\[ D_{km}(n, k) = 1 + \max((2 \log^2 k + 2 \log n), D_{km}(n/2, k)) + 2 \log n \]  
\[ \leq 1 + D_{km}(n/2, k) + 2 \log n \]  
\[ \leq 2 \log^2 n \]  

Substituting these into Equations 20 and 21 we find

\[ C(n, k) \leq 2n + 4 \frac{n}{k} \log \frac{n}{k} + 11n + 8k \log^2 k \]  
\[ D(n, k) \leq 2 \log k + 2 \log^2 \frac{n}{k} + 2 \log^2 n \]  

and minimizing these expressions with respect to \( k \), we obtain

\[ C(n, \log n) \leq 17n + 8 \log n \log^2 (\log n) = O(n), \]  
\[ D(n, \log n) \leq 2 \log \log n + 2 \log^2 n + 2 \log^2 n \]  
\[ = O(\log^2 n) \]  

It is obvious that the sorting time, \( T(n, k) \), can be similarly determined as

\[ T(n, k) = k(D_d(n, n/k) + D_s(n/k)) + D_{km}(n, k) \]  
\[ = 2k \log k + 2k \log^2 \frac{n}{k} + 2 \log^2 n \]  

and letting \( k = \log n \) gives

\[ T(n, \log n) = O(\log^3 n). \]  

As in the previous sorter case, the sorting time of this sorter can be reduced to \( O(\log^2 n) \) by using pipelining as described before.
4 Extension to Non-binary Sorting Networks

It should be easy to see that the Fish network can be used to sort arbitrary sequences simply by replacing the $n/k$-input binary sorter with an $n/k$-input general sorter, such as Batcher’s odd-even merge or bitonic sorter [Bat68,Bat90], and replacing each of the Mux-mergers by a general merger, such as Batcher’s odd-even or bitonic merger. Such a network is depicted in Figure 12 for 16 inputs, and can be easily extended to any number of inputs. Noting that an $n/k$-input Batcher’s odd-even merge sorter has $O((n/k)\log^2(n/k))$ cost and $O(\log^2(n/k))$ depth, and an $n$-input Batcher’s odd-even merger has $O(n \log n)$ cost and $O(\log n)$ depth, all in comparator level, it can be shown that the comparator-level cost, depth and sorting time of this network are given by

\[
C_g(n, k) = O(n) + O\left(\frac{n}{k} \log^2 \frac{n}{k}\right) + O(n \log n \log k) \tag{38}
\]

\[
D_g(n, k) = O(\log k) + O\left(\frac{n^2}{k}\right) + O(\log n \log k) \tag{39}
\]

\[
T_g(n, k) = O(k \log k) + O(k \frac{n^2}{k}) + O(\log n \log k) \tag{40}
\]

Replacing $k$ with $\log n$ we obtain

\[
C_g(n, \log n) = O(n \log n \log \log n) \tag{41}
\]
\[ D_2(n, \log n) = O(\log^2 n) \] \hspace{1cm} (42)
\[ T_2(n, \log n) = O(\log^3 n) \] \hspace{1cm} (43)

Again, if inputs are pipelined through the \( n/k \)-input odd-even merge sorter then
the sorting time can be reduced to \( O(\log^2 n) \) without increasing the cost beyond
\( O(n \log n \log \log n) \). It should also be noted that the constants involved in these
asymptotic expressions are in the same range as those for the binary sorter constructions
given in the earlier sections.

5 Binary Sorters as Concentrators

The concentration network problem can be defined as follows. Given \( n \) inputs and
\( m \) outputs, where \( m \leq n \), construct a network which can map any \( r \) of the \( n \) inputs
to some \( r \) distinct outputs, e.g., the first \( r \) outputs, \( 1 \leq r \leq m \). The first seminal
result on this problem was given by Pinsker [Pin73] who proved that \( O(n) \) cost
and \( O(\log n) \) depth \( n \)-input concentrators exist. Using expander graphs, Margulis
later gave an explicit construction for an \( n \)-input concentrator with \( O(n) \) cost and
\( O(\log n) \) depth, but could not specify the exact constants in the cost and depth
expressions for his construction [Mar73]. Since Margulis' construction, several
explicit constructions, most notably by Gabber and Galil, have been given where
the constants in the cost expression are known [GG81,Pip77,Alo86].

Despite their \( O(n) \) cost and \( O(\log n) \) depth, no routing schemes have been reported
for these concentrators and it is not known how hard to route them. More recently,
Koppelman and Oruç gave an \( n \)-concentrator design with \( O(n \log^2 n) \) cost, \( O(\log^2 n) \)
depth and \( O(\log^2 n) \) concentration time all in bit level [KO90]. Jan and Oruç
later improved this construction to have an \( n \)-concentrator with \( O(n \log n) \) cost,
\( O(\log n \log \log n) \) depth and \( O(\log n \log \log n) \) concentration time all in bit level [JO91].

Each of the binary sorters given in the previous section can also be used as a
concentrator without any modification. All that is needed is to assign a 0 to those
inputs to be concentrated, and assign a 1 to the remaining inputs [Cor86]. If the number of inputs to be concentrated is \( r \), then, with this assignment, a binary sorter will map the \( r \) inputs to its top \( r \) outputs. The complexities of the resulting concentrators along with those of the concentrators given in [KO90,JO91] are listed in Table 2. The other concentrators cited earlier are not shown in the table since their concentration times are not known, and also, their bit-level cost and depth can not be determined from their designs.

As can be seen in the table, the prefix and Mux-merger binary sorters match the cost of the concentrator given in [JO91] while the Fish sorter has smaller cost than both concentrators given in [KO90] and [JO91]. The depths and concentration times of our concentrators match those of the concentrators given in [KO90] and are slightly higher than those of the concentrators given in [JO91].

6 Permutation Networks Using Binary Sorters

Much work on permutation networks has been reported in the literature. The well-known Beneš network [Ben65] has \( O(n \lg n) \) cost and \( O(\lg n) \) depth while re-

Table 2
Complexities of various concentrator designs in bit level

<table>
<thead>
<tr>
<th>Construction</th>
<th>Cost</th>
<th>Depth</th>
<th>Concentration Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[KO90]</td>
<td>( O(n \lg^2 n) )</td>
<td>( O(\lg^2 n) )</td>
<td>( O(\lg^2 n) )</td>
</tr>
<tr>
<td>[JO91]</td>
<td>( O(n \lg n) )</td>
<td>( O(\lg n \lg \lg n) )</td>
<td>( O(\lg n \lg \lg n) )</td>
</tr>
<tr>
<td>Prefix sorter</td>
<td>( O(n \lg n) )</td>
<td>( O(\lg^2 n) )</td>
<td>( O(\lg^2 n) )</td>
</tr>
<tr>
<td>Mux-merger sorter</td>
<td>( O(n \lg n) )</td>
<td>( O(\lg^2 n) )</td>
<td>( O(\lg^2 n) )</td>
</tr>
<tr>
<td>Fish sorter</td>
<td>( O(n \lg \lg n) )</td>
<td>( O(\lg^2 n) )</td>
<td>( O(\lg^2 n) )</td>
</tr>
<tr>
<td>Simplified Fish sorter</td>
<td>( O(n) )</td>
<td>( O(\lg^2 n) )</td>
<td>( O(\lg^2 n) )</td>
</tr>
</tbody>
</table>
alizing arbitrary permutations on this network requires $O((\log^4 n)/(\log \log n))$ time on an $n \log n$ processor, perfect shuffle or cube-connected parallel computer [NS82]. Batcher’s sorting networks [Bat68] and those that appeared in [KO90,DO90] can also be used for permutation switching, but they require $O(n \log^3 n)$ cost and $O(\log^3 n)$ permutation time in bit-level.

More recently, Jan and Oruç proposed a permutation network with $O(n \log^2 n)$ cost and $O(\log^2 n \log \log n)$ permutation time in bit-level [JO91]. This permutation network, called a radix permuter, is recursively constructed from a distributor, two concentrators and two half-size radix permuters. Our binary sorters can be used to replace the distributor and two concentrators in this construction, since by sorting the leading bits in the destination addresses a binary sorter can distribute the inputs to the upper and lower half-size radix permuters as shown in Figure 13. If the Fish binary sorter is used in this construction then the cost $C_{rp}(n)$ and depth $D_{rp}(n)$ of this permutation network are given as

$$C_{rp}(n) = O(n \log \log n) + 2C_{rp}(n/2) = O(n \log n \log \log n)$$ (44)

$$D_{rp}(n) = O(\log^2 n) + D_{rp}(n/2) = O(\log^3 n).$$ (45)

The permutation time of this network is the same as its depth.
Table 3 lists the complexities of this permutation network along with those of other permutation networks reported earlier. The cost of the Benes network includes the cost of the $O(n \lg^2 n)$ processors in its routing model [NS82], where the $\lg n$ factor in the cost expression accounts for the bit-level cost of each processor. The table reveals that the network given in this paper has the smallest order of cost complexity and its depth and permutation time match the depths and permutation times of those networks given in [Bat68] and [KO90], and are slightly higher than the depth and permutation time of the network obtained in [JO91]. However, we should note that the binary sorters described in this paper, in particular the Muxmerger sorter, have much simpler designs and except the Fish sorter, they do not require pipelining.

### 7 Concluding Remarks

This paper introduced adaptive sorting networks, and presented an in-depth analysis of sorting binary sequences on such networks. The results are rewarding in that, under the adaptive sorting network model, we were able to obtain $O(n \lg n)$ bit-level cost binary sorting networks without pipelining, and $O(n)$ bit-level cost binary sorting networks with pipelining, and all with $O(\lg^2 n)$ bit-level sorting time.
The nonadaptive sorting network whose cost comes closest to these is the AKS binary sorting network, but the constant in its cost complexity is impractically large.

Apart from being important in their own right, binary sorters can also be used as concentrators. The binary sorters described in this paper provide the best concentrator designs known to date. Unlike the expander based concentrators [Pin73,Mar73,Pip77,GG81,Alo86], or those that use ranking trees [KO90,JO91], the binary sorters are very easy to implement in terms of constant fan-in logic gates. In particular, the Mux-merger sorter can be implemented in terms of simple $4 \times 1$ multiplexers and $1 \times 4$ demultiplexers, and does not require any comparators or adders. In addition, with its $O(n)$ bit-level cost and $O(\lg^2 n)$ bit-level routing time the simplified Fish binary sorter comes very close to being an optimal concentrator.

The paper also described the first $n$-input permutation network with $O(n \lg n)$ bit-level cost and $O(\lg^3 n)$ bit-level routing time. A permutation network with $O(n \lg^2 n)$ bit-level cost, but with a much simpler design can also be obtained by using Mux-merger sorters described in the paper.
References


