Routing Partial Permutations in Interconnection Networks based on Radix Sorting

Tripti Jain and Klaus Schneider
Department of Computer Science
TU Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—In general, sorting networks can be used as interconnection networks in that inputs are sorted according to their target addresses. However, if not all inputs need a connection to one of the outputs, partial permutations have to be implemented which cannot be directly done with sorting networks. For radix-based sorting networks, only one solution is known by Narasimha's permutation network that has unfortunately an inefficient configuration logic. In this paper, we prove that the same idea also works for more efficient ways to configure the same permutation network. We also present general alternatives for routing partial permutations in radix-based networks.

I. INTRODUCTION

Nonblocking unicast interconnection networks allow every input component to be connected to any output component that is not also the target of another input component. Mathematically speaking, such networks can implement all permutations of the components as routes through these networks. In practice, it is however often the case that not all components have to be connected to a target component, so that even all \( \sum_{i=0}^{n} i! \binom{n}{i}^2 \) partial permutations have to be implemented.

The efficient implementation of such networks turned out to be a difficult challenge for many decades. The simplest nonblocking network is the crossbar that can be implemented with \( O(n^2) \) size and \( O(\log(n)) \) depth for connecting \( n \) input components with \( n \) output components. While the depth of the crossbar is optimal, its size grows with \( O(n^2) \) and becomes quickly prohibitive for large \( n \). The challenge is therefore to develop nonblocking interconnection networks with a quasi-linear size \( O(n \log(n)^a) \) and with a poly-logarithmic depth \( O(\log(n)^b) \) (for small constants \( a, b \)).

To reduce the size of the crossbar network, Clos [8] constructed a three-stage network based on \( p \times q \) crossbar networks, and proved necessary conditions for making the network nonblocking. Based on these observations, Beneš [2], [31] constructed iterated Clos networks built by \( 2 \times 2 \) crossbars only. Beneš networks have optimal size \( O(n \log(n)) \) and optimal depth \( O(\log(n)) \). However, computing the right configurations of the \( 2 \times 2 \) crossbars turned out to be difficult and required a circuit depth larger than the network itself [25], [21]. Many simpler networks based on \( 2 \times 2 \) crossbars were then proposed, e.g., the \( \Omega \)-network [20], the butterfly (Banyan) networks [9], fat trees [23], flattened butterfly [18], the 2-dilated butterfly network [30] to name just a few. All of these networks are however blocking, i.e., they can only implement a small fraction of the possible permutations.

The use of sorting networks [1] is therefore an attractive alternative for the design of nonblocking interconnection networks: The inputs are simply sorted according to their target addresses by compare-and-swap modules that require no additional configuration logic. However, the implementation of partial permutations by sorting networks is not that simple and depends on the used sorting algorithms as we will outline in the next section. In particular, the Batcher-Banyan network is a general solution for merge-based networks, while for radix-based sorting networks that were often preferred for interconnection networks [11], [22], [6], [5], [19], [27], [12], only one solution was known so far that assumes a special network [27] due to Narasimha. Unfortunately, the configuration logic of the Split modules (see Figure 1) of this network has a bad circuit depth of \( O(n) \).

In this paper, we significantly improve Narasimha's network in that we present two alternative circuits to derive the same configuration of Split modules with a circuit of size \( O(n) \) and depth \( O(\log(n)) \). We moreover prove that a class of configuration algorithms that we call prefix-defined configurations will correctly implement all partial permutations with Narasimha's construction using a front-end concentrator, and thus also our optimized versions can be extended to partial permutations in the same way.

The outline of the paper is as follows: In the next section, we report about related work on the use of sorting networks as interconnection networks including known solutions for partial permutations. Section III-A presents the permutation network used by Narasimha in [27], and Sections III-B, III-C, and III-D present three different circuits to configure that network. While the first two circuits were published in [27] and [12], the third one is a first contribution of this paper. Section IV contains a second contribution: We prove that all three networks can be extended to route partial permutations. Furthermore, we also discuss alternatives for implementing partial permutations by implementing ternary Split modules. Finally, Section V shows by experimental results that our new circuits significantly improve the original circuit.
II. SORTING NETWORKS AS INTERCONNECTION NETWORKS

As already mentioned, sorting networks are an interesting way to implement nonblocking interconnection networks. These sorting networks are built by compare-and-swap modules that compare the target addresses of two inputs and swaps them if needed to generate the two outputs. This way, these networks are self-routing, i.e., do not require additional configuration logic.

There are two important classes of such sorting networks, namely the merge-based (MBS) and the radix-based (RBS) sorting networks which can be recursively defined as shown in Figure 1. In the merge-based approach, a sorting network MBS(n) for n inputs is recursively constructed by splitting the given sequence into two halves, recursively sorting these by sorting networks MBS(n/2) of half the size, and then merging the sorted halves by a merge module Merge(n). Well-known sorting networks following this paradigm are Batcher’s bitonic and odd-even sorting networks [1] and related ones [28].

In radix-based sorting networks, the given inputs are partitioned into two halves by a Split(n) module, e.g., by sorting them according to the most significant bit of their target address. Thus, after the Split(n) module, the given inputs have already been routed to the right halves, so that the remaining problems can be recursively dealt with in the same way (ignoring now the most significant bits of the target addresses).

The implementation of radix-based sorting networks is essentially given by the implementation of the Split(n) modules. Their size and depth determines the size and depth of the entire network. There are many ways to implement a Split(n) module, e.g., by means of concentrators [29, 7, 24]: A (n, m)-concentrator is a circuit with n inputs and m \( \leq n \) outputs that can route any given number \( k \leq m \) of valid inputs to \( k \) of its \( m \) outputs. Split modules can therefore be implemented by two \((n, \frac{n}{2})\)-concentrators: One that routes the \( \frac{n}{2} \) inputs with a most significant target address bit 1 from the \( n \) inputs to the \( \frac{n}{2} \) outputs, and another one routing the other \( \frac{n}{2} \) inputs with a most significant target address bit 0 to its outputs. The two concentrators' outputs are then the upper and lower halves of the Split module's outputs. In many practical cases like [11, 22, 5, 19, 27, 12], however, binary sorters were used as Split module even though we know that we can do better [15, 13] (see also Section IV-D).

Independent of the choice of a particular sorting algorithm, sorting networks at first only implement total permutations in that they can sort the \( n \) inputs by their target addresses as long as these are numbers 0, . . . , \( n-1 \). However, if some source components do not need a connection to a target address, their target addresses are invalid, denoted as \( \bot \) in the following.

For merge-based sorting networks, there is a well-known solution known as the Batcher-Banyan network [10], [26]. The main idea is to treat \( \bot \) as a number larger than all target addresses so that after using a sorting network this way, one obtains an output sequence \( y_0, \ldots, y_{k-1}, y_k, \ldots, y_{n-1} \) where the \( k \) valid inputs form the sorted prefix \( y_0 \leq \ldots \leq y_{k-1} \) while the invalid ones are placed in the suffix \( y_{k}, \ldots, y_{n-1} \). A final (Banyan) permutation network can then be used to move the invalid outputs \( y_{k}, \ldots, y_{n-1} \) inside the prefix of the valid ones \( y_0, \ldots, y_{k-1} \) such that the valid ones will finally appear at the right places.

The same approach does however not work for the radix-based networks: If an input sequence with invalid target addresses is given there, it is unclear in the Split modules whether invalid inputs should be routed to the upper or lower subnetworks. To determine that, one would have to know the number of valid inputs having a most significant bit 0 and 1. However, computing these numbers requires a substantial effort with a circuit of minimal depth \( O(\log(n)) \). Alternatively, the binary sorters used as Split modules must become ternary sorters where inputs must be sorted as 0 \( \leq \bot \leq 1 \) using their most significant target address bits before splitting the sequences sorted this way into halves. It is however not at all clear how the known binary sorters [11, 22, 5, 19, 27, 12] could be extended this way (unless using two of these modules doubling the size of the circuits as discussed in Sections IV-B and IV-C).

Narasimha addressed this problem in [27] where he first introduced the recursively defined permutation network shown in Figure 2. He then proved that the switches of this network can be configured so that it will act as a binary sorter and can be used as such as Split module for RBS networks. Unfortunately, the configuration logic he suggested leads to a circuit depth of \( O(n) \). We review and optimize his work in [12] and in the next section. Moreover, he claimed that a further Split module (called a front-end concentrator) for his RBS network (as shown in Figure 3) will allow one to route even partial permutations. However, he neither gave a proof of this claim, nor did he discuss how that construction could be generalized to other networks. In essence, he used an additional Split module in the front of the radix-based sorting network whose output sequence \( y_0, \ldots, y_{k-1}, y_k, \ldots, y_{n-1} \) consists of the valid inputs \( y_0, \ldots, y_{k-1} \) (however not sorted) as a prefix, and the invalid ones \( y_k, \ldots, y_{n-1} \) as a suffix. With this pre-computation, Narasimha claimed that his network will also correctly deal with partial permutations. We give a formal proof of this statement in Section IV and generalize it to an entire class of configuration circuit.
III. DISTRIBUTION-BASED RADIX-SORTING INTERCONNECTION NETWORK

In this section, we review Narasimha’s construction of an interconnection network based on radix-based sorting. We first consider only total permutations and will consider partial permutations in Section IV. In Section III-A, we first introduce the Split module’s permutation network and prove how it should be configured to act as a binary sorter. We then discuss three circuits to implement this configuration logic in Sections III-B, III-C, and III-D.

A. The Split Module’s Permutation Network (RB-FS-RV)

All three networks discussed in this section are based on a reverse-banyan flip-shuffle permutation network with an outgoing permutation that reverses the addresses. This network is recursively constructed as shown in Figure 2 for $n$ binary inputs $x_0, \ldots, x_{n-1}$. As can be seen, it starts with a column of $2$ crossbar switches that are controlled by inputs $p_0, \ldots, p_{n-2}$. Switch $i$ is thereby in one of two modes: If the configuration input $p_i$ is 0, it is in ‘through’ mode, thus mapping its inputs $x_{2i}$ and $x_{2i+1}$ to its outputs $u_{2i}$ and $u_{2i+1}$, respectively, and if the configuration $p_i$ is 1, it is in ‘crossed’ mode, thus mapping its inputs $x_{2i}$ and $x_{2i+1}$ to its outputs $u_{2i+1}$ and $u_{2i}$, respectively. Note that the outputs $u_{2i}$ and $u_{2i+1}$ of switch $i$ are then mapped to the lower and upper subnetwork’s input $i$, respectively (where the lower and upper one’s local addresses $0, \ldots, \frac{n}{2} - 1$ are associated with the global addresses $0, \ldots, \frac{n}{2} - 1$ and $\frac{n}{2}, \ldots, n - 1$, respectively).

It can be directly seen in Figure 2 that the permutation between the columns of the switches is a flip-shuffle permutation $\sigma$ that maps an address with the address bits $a_{n-1}, \ldots, a_0$ to $a_0, a_{n-1}, \ldots, a_1$. The output permutation shown in Figure 2 is the perfect shuffle permutation $\pi$ that maps the address bits $a_{n-1}, \ldots, a_0$ to $a_{p-2}, \ldots, a_0, a_p-1$. At the end, we obtain a reverse banyan network with flip-shuffle $fs$ permutations between its stages, and a reverse-bit $rv$ output permutation that maps address bits $a_{p-1}, \ldots, a_0$ to $a_0, \ldots, a_{p-1}$ (see [14], [12]). A complete RBS-network based on this permutation network is shown in Figure 3 when we ignore the green shaded part. After the green-shaded part, which will be explained in Section IV, we can see Split modules for $n = 16, 8, 4, 2$ inputs, and all of them are constructed according to Figure 2.

As we will prove below, the switches should be configured such that the 1s (and 0s) of the binary input sequence $x_0, \ldots, x_{n-1}$ are equally distributed to the two subnetworks, and in case that the number of 1s (and 0s) is odd, then the additional 1 is routed to the upper subnetwork (and the additional 0 is routed to the lower subnetwork). For this reason, the final output permutation that routes outputs $i$ of the lower and upper subnetworks to the final outputs $y_0, y_1, \ldots, y_{n-1}$, respectively, will shuffle the sorted binary sequences coming from the subnetworks into a single binary sorted sequence. Narasimha [27] showed that using parities $p_i = x_0 \oplus \ldots \oplus x_{n-1}$, the switches will make the Split module a binary sorter [12]. For this reason, we obtained the following theorem (see also [12]):

**Theorem 1:** The reverse-banyan flip-shuffle permutation network with outgoing reverse permutation as recursively constructed as shown in Figure 2 becomes a binary sorter if all switch configurations $p_i$ are computed as parities $p_i = x_0 \oplus \ldots \oplus x_{n-1}$ of the input prefixes $x_0, \ldots, x_{n-1}$.

**Proof:** As already discussed, the overall idea is to distribute the 1s (and therefore also the 0s) of any prefix of the inputs $x_0, \ldots, x_i$ equally to the two subnetworks, and in case the number of 1s is odd, we route the additional 1 to the upper and the additional 0 to the lower subnetwork. We first prove that this can be achieved by configuring the switches by the parities $p_i = x_0 \oplus \ldots \oplus x_{n-1}$ of the input:

- If $x_{2i} = x_{2i+1}$ holds, the configuration of switch $i$ does not matter since then either two 0s or two 1s are routed from this switch to the two subnetworks which does not change the equal distribution.
- If $x_{2i} \neq x_{2i+1}$ holds, one of the two is 0 while the other one is 1. Depending on the number of 1s that occurred in the prefix $x_0, \ldots, x_{2i-1}$ so far, the new 1 has to be routed to the lower or upper subnetwork so that our equal distribution of 1s is maintained. To this end, assume that $u_{p_0}$ and $l_{p_1}$ denote the numbers of 1s occurring in $x_0, \ldots, x_{2i-1}$ that were already sent to the upper and lower subnetworks, respectively. Since we either have $u_{p_1} = l_{p_1}$ or $u_{p_1} = l_{p_1} + 1$ as induction hypothesis, depending on whether the number of 1s was even or odd, respectively, we just need to know whether the number of 1s in $x_0, \ldots, x_{2i-1}$ is even or odd. Now note that $p_{i+1} \oplus x_{2i+1} = x_0 \oplus \ldots \oplus x_{2i-1}$ holds (since $x \oplus 0 = x$ and $x \oplus 0 = x$ holds). Hence, we have the following:
  - $p_{i+1} = l_{p_1}$ if and only if $p_i \oplus x_{2i} = 0$ holds.
  - $p_{i+1} = l_{p_1} + 1$ if and only if $p_i \oplus x_{2i} = 1$ holds.

Inspecting the cases shown in Table I finally shows that using $p_i := x_0 \oplus \ldots \oplus x_{n-1}$ as configuration of switch $i$ will equally distribute the 1s in $x_0, \ldots, x_{2i+1}$ to the two subnetworks.
Hence, using the parities \( p_i := x_0 \oplus \ldots \oplus x_2i \) as configuration of the switches, the 1s contained in the input sequence \( x_0, \ldots, x_{n-1} \) are equally distributed to the two subnetworks in case their number is even, and otherwise the additional 1 is sent to the upper and the additional 0 is sent to the lower subnetwork. Assuming the equal distribution, we finally prove the correctness of the binary sorter by induction on the number \( n \) of inputs/outputs:

- **induction base**: For \( n = 2 \), the network consists of a single \( 2 \times 2 \) crossbar switch. If we use for such a single crossbar switch \( p_0 \equiv x_0 \) as its configuration, it can be easily seen by Table I that it will sort its two binary inputs. This already proves the induction base.

- **induction step**: In the induction step, we may assume by the induction hypothesis that the two subnetworks in Figure 2 implement binary sorters. Now, it is important that we have equally distributed the 1s of the input sequence \( x_0, \ldots, x_{n-1} \) to the two subnetworks, and in case their number was odd, the additional 1 is in the upper network. Because of this, the final perfect shuffle permutation will produce a sorted sequence of the two sorted subsequences of the local subnetworks. Note that the sequences generated by the subnetworks are either both \( 0^i1^{\frac{n}{2}-i} \) (if the same number of 0s/1s were sent to both subnetworks), and otherwise \( 0^i1^{\frac{n}{2}-i} \) from the upper subnetwork.

### B. Configuration I: Sequential Parity Computation

In the previous section, we have seen that the configuration of switch \( i \) should be computed as \( p_i := x_0 \oplus \ldots \oplus x_{2i} \). The simplest circuit to implement this configuration logic has been suggested by Narasimha in [27] where the switch modules are extended by a further input \( p_{\text{in}} \) and a further output \( p_{\text{out}} \). In switch \( i \), we assume that \( p_{\text{in}} := x_0 \oplus \ldots \oplus x_{2i-1} \) and in switch \( i \), we assume \( p_{\text{in}} := 0 \). We can therefore simply compute in switch \( i \):

\[
\begin{align*}
 p_i & := p_{\text{in}} \oplus x_{2i} \\
 p_{\text{out}} & := p_i \oplus x_{2i+1}
\end{align*}
\]

so that we obviously have \( p_i = x_0 \oplus \ldots \oplus x_{2i} \) as required by Theorem 1 and \( p_{\text{out}} = x_0 \oplus \ldots \oplus x_{2i+1} \) as assumed by switch \( i+1 \).

Since we only need two additional XOR gates for each switch, the size of the obtained circuit is very good: It leads to a circuit size of \( O(n \log(n)) \) for the \textit{Split} modules. Unfortunately, their depth grows by \( O(n) \) which is quite slow compared to other networks (as the ones in the next sections).

### C. Configuration II: Parallel Parity Computation

In [12], we presented a circuit to efficiently compute the switch configurations \( p_i = x_0 \oplus \ldots \oplus x_{2i} \) as required by Theorem 1. The core of that circuit is based on a work-efficient parallel prefix computation [17] with an additional pre-processing step.

The parallel parity computation can determine all \( p_i \) for \( i \in \{0, \ldots, \frac{n}{2} - 1\} \) of one column as follows:

1. Compute the following list using \( \frac{n}{2} \) XOR gates in one step: \( [b_0, \ldots, b_{\frac{n}{2}-1}] = [a_0 \oplus a_1, a_2 \oplus a_3, \ldots, a_{n-2} \oplus a_{n-1}] \), i.e., we define \( b_i := a_{2i} \oplus a_{2i+1} \) for \( i = 0, \ldots, \frac{n}{2} - 1 \).

2. Recursively apply the parallel prefix computation to the list \( [b_0, \ldots, b_{\frac{n}{2}-1}] \) and obtain its prefixes \( [c_0, \ldots, c_{\frac{n}{2}-1}] \). Thus, we have \( c_j := b_0 \oplus \ldots \oplus b_j \) for \( j = 0, \ldots, \frac{n}{2} - 1 \).

3. Since \( b_j := c_{2j} \oplus c_{2j+1} \) holds, we therefore have \( c_j := a_0 \oplus a_1 \oplus \ldots \oplus a_{2j} \ominus a_{2j+1} \), and have therefore already the prefixes \( P_{2j+1} = c_j \) for the odd indices \( 2j + 1 \) for \( j = 0, \ldots, \frac{n}{2} - 1 \). The prefixes of the even indices \( P_{2j} \) are now computed in one parallel step as \( P_0 := a_0 \) and \( P_{2j} := P_{2j-1} \oplus 0 = c_{j-1} \oplus a_{2j} \) for \( j = 1, \ldots, \frac{n}{2} - 1 \) using \( \frac{n}{2} \) XOR gates.

The correctness of the above algorithm is easily shown by induction, and as shown in [12], we obtain the following depth and size for a circuit:

\[
\begin{align*}
 D_{\text{prefix}}(n) & = 2 \log(n) - 2 \text{ for } n > 2 \\
 S_{\text{prefix}}(n) & = 2n - \log(n) - 2 \text{ for all } n = 2^c
\end{align*}
\]

As outlined in [12], we do not need all parities that the above general parallel prefix sum algorithm would compute. Instead, we only need those parities \( p_i := x_0 \oplus \ldots \oplus x_{2i} \) for \( i = 0, \ldots, \frac{n}{2} - 1 \) ending in \( x_{2i} \). We can optimize the usual algorithm as follows: We first compute the following values \( z_i \) for \( i = 0, \ldots, \frac{n}{2} - 1 \):

\[
z_i := \begin{cases} x_0 & \text{for } i = 0 \\ x_{2i-1} \oplus x_{2i} & \text{for } i = 1, \ldots, \frac{n}{2} - 1 \end{cases}
\]

The computation of all \( z_i \) can be done in one step using exactly \( \frac{n}{2} - 1 \) XOR gates. After this, we apply the parallel prefix computation mentioned above to the \( \frac{n}{2} \) values \( z_0, \ldots, z_{\frac{n}{2}-1} \). The result values \( p_0, \ldots, p_{\frac{n}{2}-1} \) are then our values needed to configure the switches because of the following:

\[
p_i := z_0 \oplus z_1 \oplus \ldots \oplus z_i = x_0 \oplus (x_1 \oplus x_2) \oplus \ldots \oplus (x_{2i-1} \oplus x_{2i})
\]

Hence, we can compute all \( p_i \) for \( i \in \{0, \ldots, \frac{n}{2} - 1\} \) of one column with a circuit having the following depth \( D_{\text{colCF}}(n) \) and size \( S_{\text{colCF}}(n) \) for \( n > 1 \):

\[
\begin{align*}
 D_{\text{colCF}}(n) & = 2 \log(n) - 3 \text{ for } n > 4 \\
 S_{\text{colCF}}(n) & = \frac{3}{2}n - \log(n) - 2 \text{ for } n > 1
\end{align*}
\]

This computation has to be done recursively for all columns of the \textit{Split} module shown in Figure 2. Thus, a \textit{Split} module of depth \( D_{\text{node}}(n) = \log(n) - \log(n-1) + 2 \) and \( S_{\text{node}}(n) = (q + \log(n) + 3) \frac{3}{2} \log(n) + \log(n) - \frac{3}{2}q + 4 \) will be obtained where \( q \) is the number of message bits.
D. Configuration III: Ranking-based Configuration

The parallel computation of the parities to configure the switches shown in the previous section is already a big improvement of the depth of the circuit without compromising its size (see Table III). However, we have to compute the parities of each column in the permutation network shown in Figure 2 by a separate parallel prefix computation which therefore ends up with an overall circuit depth of \(O(\log(n)^2)\) for the Split module\(^2\) and \(O(\log(n)^3)\) for the entire network.

It is therefore natural to ask whether the obtained depth is optimal. In this section, we show that this is not the case in that we present another circuit to configure the switches that leads to a depth of \(O(\log(n))\) for the binary sorter and \(O(\log(n)^2)\) for the entire network. This circuit is based on the computation of ranks as suggested by [19] for the generalized cube network. Our contribution of this section is therefore to show that a ranking-based approach can also configure the network of Figure 2. To this end, we first compute the following ranks with the most significant bit msb of switch \(x_i\):

\[
r_i := \left(\sum_{j=0}^{i} -\text{msb}(x_j)\right) - 1 \quad \text{for} \quad i = 0, \ldots, n-1,
\]

i.e., the number of inputs \(x_j\) in the prefix \(x_0, \ldots, x_i\) that have a most significant bit \(\text{msb}(x_i) = 0\) minus 1. These ranks can be computed as shown in [19], but also with a parallel prefix computation similar to the one of the parities we used in [12] (using adders instead of XOR gates). The ranks can now be used as local addresses of the network of Figure 2 in that an input \(x_i\) with \(\text{msb}(x_i) = 0\) must be routed to output \(y_r\). We do not care about inputs \(x_i\) with \(\text{msb}(x_i) = 1\), since routing the inputs \(x_i\) with \(\text{msb}(x_i) = 0\) to the correct places will implicitly also route the others in the right part of the output sequence (maybe in a permuted ordering). This will make sure that the Split module sorts its inputs \(x_i\) by \(\text{msb}(x_i)\).

In general, permutation networks like the considered RB-FS-RV network are blocking, i.e., cannot implement all permutations of input addresses to target addresses. However, the RB-FS-RV network is able to route the 0s to monotonically increasing target addresses as proved in the following. We will also see that the ranking-based approach will compute exactly the same configuration as the approaches using parity bits. To this end, consider Table II where we determine the configuration \(p_i\) of switch \(i\) based on the most significant bits \(\text{msb}(x_2i)\) and \(\text{msb}(x_2i+1)\) of the target addresses of inputs \(x_{2i}\) and \(x_{2i+1}\), respectively, and their ranks \(r_{2i}\) and \(r_{2i+1}\), respectively.

The configuration \(p_i\) of switch \(i\) is obtained according to Table II as follows: First note that the lower subnetwork in Figure 2 is connected with the even output addresses, and that the upper subnetwork is connected with the odd output addresses. Second, recall that rank \(r_i = k\) means that there are \(k+1\) 0s in the prefix \(x_0, \ldots, x_k\). Thus, we have the following cases (note that we do not care about routing inputs \(x_i\) with \(\text{msb}(x_i) = 1\) since these are implicitly routed to the remaining target addresses):

- If \(\text{msb}(x_{2i}) = \text{msb}(x_{2i+1}) = 0\) and \(r_{2i} = 2k + 1\) holds, then \(r_{2i+1} = r_{2i} + 1 = 2k + 2\) holds, and the prefix \(x_0, \ldots, x_{2i-1}\) contains \((r_{2i} - 1) + 1 = 2k\) 0s which are equally distributed, i.e., \(k\) of them have been sent to the lower subnetwork and \(k\) others to the upper subnetwork that occupy the local addresses \(0, \ldots, k - 1\) there. The configuration \(p_i\) does not matter here, we will send one input to the lower and the other to the upper subnetwork with local address/rank \(k = \left\lfloor \frac{2k}{2} \right\rfloor = \left\lfloor \frac{r_{2i} - 1}{2} \right\rfloor\).
- If \(\text{msb}(x_{2i}) = \text{msb}(x_{2i+1}) = 0\) and \(r_{2i} = 2k + 1\) holds, then \(r_{2i+1} = r_{2i} + 1 = 2k + 2\) holds, and the prefix \(x_0, \ldots, x_{2i-1}\) contains \((r_{2i} - 1) + 1 = 2k + 1\) 0s which are equally distributed, i.e., \(k + 1\) of them have been sent to the lower subnetwork and \(k\) others to the upper subnetwork that occupy the local addresses \(0, \ldots, k\) and \(0, \ldots, k - 1\) there. The configuration \(p_i\) does matter here in the sense that we will send 0s to each subnetwork, but we have to define \(p_i := 1\) so that \(x_{2i}\) will be sent to the upper subnetwork with local address/rank \(k = \left\lfloor \frac{2k + 1}{2} \right\rfloor = \left\lfloor \frac{r_{2i} - 1}{2} \right\rfloor\) while \(x_{2i+1}\) will be sent to the lower subnetwork with local address/rank \(k + 1 = \left\lfloor \frac{2k + 2}{2} \right\rfloor = \left\lfloor \frac{r_{2i} - 1}{2} \right\rfloor + 1\).
- If \(\text{msb}(x_{2i}) = 0\) and \(\text{msb}(x_{2i+1}) = 1\) and \(r_{2i} = 2k + 1\) holds, then the prefix \(x_0, \ldots, x_{2i-1}\) contains \((r_{2i} - 1) + 1 = 2k\) 0s which are equally distributed, i.e., \(k\) of them have been sent to the lower subnetwork and \(k\) others to the upper subnetwork that occupy the local addresses \(0, \ldots, k - 1\) there. The configuration \(p_i\) does not matter here, we will send 0s to each subnetwork. We have to define \(p_i := 0\) so that \(x_{2i}\) will be sent to the lower subnetwork with local address/rank \(k = \left\lfloor \frac{2k}{2} \right\rfloor = \left\lfloor \frac{r_{2i} - 1}{2} \right\rfloor\) while \(x_{2i+1}\) will be sent to the upper subnetwork (and we don't care about its address).
- If \(\text{msb}(x_{2i}) = 0\) and \(\text{msb}(x_{2i+1}) = 1\) and \(r_{2i} = 2k + 1\) holds, then the prefix \(x_0, \ldots, x_{2i-1}\) contains \((r_{2i} - 1) + 1 = 2k\) 0s which are equally distributed, i.e., \(k\) of them have been sent to the lower subnetwork and \(k\) others to the upper subnetwork that occupy the local addresses \(0, \ldots, k - 1\) there. The configuration \(p_i := 1\) so that \(x_{2i}\) will be sent to the upper subnetwork with local address/rank \(k = \left\lfloor \frac{2k}{2} \right\rfloor = \left\lfloor \frac{r_{2i} - 1}{2} \right\rfloor\) while \(x_{2i+1}\) will be sent to the lower subnetwork (and we don't care about its address).

\(^2\)Since there are \(\log(n)\) columns, we need \(\log(n)\) many such computations with sizes \(n, \frac{n}{2}, \frac{n}{4}, \ldots, 2\).
have been sent to the lower subnetwork and \( k \) others to the upper subnetwork that occupy the local addresses \( 0, \ldots, k-1 \) there. We define \( p_i := 1 \) so that \( x_{2i+1} \) will be sent to the lower subnetwork with local address/rank \( k = \left\lfloor \frac{2i+1}{2} \right\rfloor = \frac{k}{2} \) while \( x_{2i} \) will be sent to the upper subnetwork (and we don’t care about its address).

- If \( \text{msb}(x_{2i}) = 1 \), \( \text{msb}(x_{2i+1}) = 0 \), and \( r_{2i+1} = 2k + 1 \) holds, then the prefix \( x_{0i}, \ldots, x_{2i-1} \) contains \( (r_{2i+1} - 1) + 1 = 2k + 1 \) 0s which are equally distributed, i.e., \( k + 1 \) of them have been sent to the lower subnetwork and \( k \) others to the upper subnetwork that occupy the local addresses \( 0, \ldots, k \) and \( 0, \ldots, k-1 \) there. We define \( p_i := 0 \) so that \( x_{2i+1} \) will be sent to the upper subnetwork with local address/rank \( k = \left\lfloor \frac{2i+1}{2} \right\rfloor = \frac{k}{2} \) while \( x_{2i} \) will be sent to the lower subnetwork (and we don’t care about its address).
- The only not yet considered cases are those where \( \text{msb}(x_{2i}) = \text{msb}(x_{2i+1}) = 1 \) hold with arbitrary ranks. Since we do not care about routing these inputs, the configuration \( p_i \) can be chosen arbitrarily in these cases, and we also do not have to determine local addresses/ranks for these inputs.

Note that the above discussed cases which are also listed in Table II are complete, since in the first two cases, \( r_{2i+1} \) is determined by the values of \( \text{msb}(x_{2i}), \text{msb}(x_{2i+1}) \), \( r_{2i} \).

For this reason, we now derive the following Karnaugh-Veitch diagram from Table II using the least significant bits \( \text{lsb}(r_{2i}), \text{lsb}(r_{2i+1}) \) of \( r_{2i}, r_{2i+1} \), respectively:

<table>
<thead>
<tr>
<th>( \text{msb}(x_{2i}) ) ( \text{lsb}(r_{2i}) )</th>
<th>( \text{msb}(x_{2i+1}) ) ( \text{lsb}(r_{2i+1}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

We therefore can define \( p_i \) by the following minimal disjunctive normal forms:

\[
\begin{align*}
  p_i := & \quad \text{msb}(x_{2i}) \land \text{lsb}(r_{2i}) \lor \neg \text{msb}(x_{2i}) \land \text{lsb}(r_{2i}) \\
  p_i := & \quad \text{msb}(x_{2i+1}) \land \text{lsb}(r_{2i}) \lor \neg \text{msb}(x_{2i+1}) \land \text{lsb}(r_{2i+1}) \\
  p_i := & \quad \neg \text{msb}(x_{2i}) \land \text{lsb}(r_{2i}) \lor \neg \text{msb}(x_{2i+1}) \land \text{lsb}(r_{2i+1}) \\
  p_i := & \quad \text{if } \text{msb}(x_{2i}) \text{ then } \neg \text{lsb}(r_{2i+1}) \text{ else } \text{lsb}(r_{2i}) \\
  p_i := & \quad \text{if } \text{msb}(x_{2i+1}) \text{ then } \text{lsb}(r_{2i}) \text{ else } \neg \text{lsb}(r_{2i+1})
\end{align*}
\]

Using multiplexers, we can also equivalently define

\[
\begin{align*}
  p_i := & \quad \text{if } \text{msb}(x_{2i}) \text{ then } \neg \text{lsb}(r_{2i+1}) \text{ else } \text{lsb}(r_{2i}) \\
  p_i := & \quad \text{if } \text{msb}(x_{2i+1}) \text{ then } \text{lsb}(r_{2i}) \text{ else } \neg \text{lsb}(r_{2i+1})
\end{align*}
\]

Having determined the configurations \( p_i \) of switch \( i \) in the first column of the network shown in Figure 2, we can recursively determine the configurations of the other columns in the subnetworks since we have also determined the local addresses/ranks \( rL_i \) and \( rU_i \) to be used in the subnetworks as shown in Table II: As can be seen, the local address/rank \( rL_i \) and \( rU_i \) is simply obtained by removing the least significant bit from the rank of the value that is routed to that place.

Using ranks as local target addresses for the Split modules determines the same configurations that we obtained by computing parity bits: Note that all inputs \( x_i \) with \( \text{msb}(x_i) = 0 \) were routed to the lower subnetwork if their rank was even, and otherwise to the upper subnetwork. Hence, also the ranking-based approach equally distributes the inputs \( x_i \) with \( \text{msb}(x_i) = 0 \) and \( \text{msb}(x_i) = 1 \), respectively, to the lower and upper subnetworks, given preference to the lower and upper subnetworks for additional inputs \( x_i \) with \( \text{msb}(x_i) = 0 \) and \( \text{msb}(x_i) = 1 \), respectively.

It can be easily seen that size and depth of this circuit to compute the configuration logic is asymptotically better than the circuits presented in the previous sections (and even optimal!): Note that the ranks can be computed by a parallel prefix sum using \( O(n) \) gates and \( O(\log(n)) \) depth\(^3\). However, while the ranking-based approach improves the size and depth of the configuration logic, it has the disadvantage that the computed ranks now have to be forwarded to the subnetworks which requires additional switches. Thus, while saving gates for computing the configuration, we have to add gates for forwarding the centrally computed configuration (the ranks) through the network.

While the final size of the entire RBS network is asymptotically the same for all three circuits, namely \( O(n \log(n)^3) \), the depth of the RBS network with the ranking-based approach is only \( O(\log(n)^2) \) while it is \( O(\log(n)^3) \) for the parallel parity computation, and \( O(n) \) for the sequential parity computation [12]. However, we have seen by our experiments, that the circuit size of the ranking-based approach is not as good as the one using the parallel parity computation (see Table III).

### IV. Routing Partial Permutations

In the previous section, we considered circuits of different depths and sizes to compute configurations of the RB-FS-RV permutation network such that it acts as a binary sorter to be used as Split module of an RBS network. In Section IV-A, we will first prove as a second contribution of this paper that all of these three networks can be easily generalized to deal with invalid inputs, i.e., partial permutations, that arise if some inputs may not have to be connected to some outputs. In Sections IV-B and IV-C, we consider less efficient, but more general alternatives using ternary sorts instead of binary sorts, and Section IV-D presents a general optimization of Split modules.

\[
\begin{array}{c|c|c|c}
  x_{i,0} & \ldots & x_{i,q-1} & \ldots x_{i,q+1} \ldots x_{i,q+\log(n)} \\
\end{array}
\]

For all networks with partial permutations, we assume that any input \( x_i \) is a bi-vector in the format given above: The leftmost \( q \) bits \( x_{i,0}, \ldots x_{i,q-1} \) are the message bits that should be sent to an output, \( x_{i,q} \) is the valid bit that indicates whether this input contains a message and shall be connected to some output, and the remaining bits \( x_{i,q+1} \ldots x_{i,q+\log(n)} \) are the bits of the target address where \( x_{i,q+\log(n)} \) is the most significant bit.

For partial permutations, the valid bit \( x_{i,q} \) and bit \( x_{i,q+\log(n)} \) are now considered together as most significant

\(^3\)Even though the adders have to add numbers with more than one bit, the circuit obtained by these (carry-ripple) adders has a depth of \( O(\log(n)) \) since the additions of the next higher level can already start their work when the first bit of the lower level is available.
bit of input $x_i$ in that we define the following value $\text{msb}(x_i) \in \{0, \bot, 1\}$ by $x_{i,q}$ and $x_{i,q+\log(n)}$ as follows:

<table>
<thead>
<tr>
<th>$x_{i,q}$</th>
<th>$x_{i,q+\log(n)}$</th>
<th>$\text{msb}(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We will therefore consider now inputs $x_i$ of the Split modules that have $\text{msb}(x_i) \in \{0, \bot, 1\}$ where $\bot$ means that the $x_i$ has no valid target address, while 0 and 1 are the most significant bits of the otherwise valid target address. In every stage of the RBS networks, the Split modules will remove a target address bit $x_{i,q+\log(n)}$ in the RBS network so that finally only the message and the valid bit will arrive at the output.

A. Front-end Concentrators for Prefix-Defined Networks

A major observation made by Narasimha [27] was that his RBS network also works for partial permutations if an additional Split module (called a front-end concentrator) is added (see Figure 4 and 5 for the general construction and Figure 3 for $n = 16$ inputs). Such a front-end concentrator is also a RB-FS-RV network that is used as a special binary sorter: To this end, we simply sort the inputs $x_i$ to their negated valid bits $\neg x_{i,q}$. This means that the invalid messages are sorted to a suffix of the sequence. As a result, the vector $u_i$ that is generated as output from the front-end concentrator is a prefix sequence defined as follows:

**Definition 1 (Prefix Sequence):** Sequence $x_0, \ldots, x_{n-1}$ is called a prefix sequence, if the sequence $\text{msb}(x_0), \ldots, \text{msb}(x_{n-1})$ of their most significant bits is of the form $b_0, \ldots, b_k, \bot, \ldots, \bot$ with $b_i \in \{0, 1\}$.

The task of the front-end concentrator is therefore to partition the inputs into valid and invalid ones which can be easily done by sorting according to $\neg x_{i,q}$. We will prove next that the configurations we determined in the previous section can route all prefix sequences correctly. This is easily seen by the ranking-based approach, so that we first focus on this one: After the front-end concentrator, we apply the following mapping before we forward the inputs to the Split modules:

Thus, we define $\rho_{\bot}(x_{i,q}) := x_{i,q}$ and $\rho_{\bot}(x_{i,q+\log(n)}) := x_{i,q} \land x_{i,q+\log(n)}$ which will make sure that bit $x_{i,q+\log(n)}$ is only 1 for valid entries. Ranks are now computed as follows:

$$r_i := \left(\sum_{j=0}^{i} -\rho_{\bot}(x_{j,q+\log(n)})\right) - 1 \text{ for } i = 0, \ldots, n-1,$$

i.e., we are computing ranks for those inputs that are either invalid $\neg x_{i,q}$ or have $x_{i,q+\log(n)} = 0$. These inputs are then routed by the configuration we determined in Section III-D to the right local target addresses, i.e., their ranks. Since we already have proved that this is always possible for any binary sequence, we conclude the following theorem:

**Theorem 2 (Ternary Sorting Prefix Sequences):** All binary sorters configured by the circuits described in Sections III-B, III-C, and III-D can sort any ternary prefix sequence by the ordering $0 \preceq \bot \preceq 1$.

Proof: As we compute the ranks as explained above, inputs with $\text{msb}(x_i) \in \{0, \bot\}$ will be routed correctly to the local addresses given as their ranks. As these are the first contiguous addresses $0, \ldots, k$, and are monotonically increasing, i.e., $i \leq j$ implies $r_i \leq r_j$, it follows that inputs $x_i$ with $\text{msb}(x_i) = 0$ have smaller ranks than inputs $x_i$ with $\text{msb}(x_i) = \bot$. Thus, all $0$s will appear in a prefix of the sequence before the $\bot$s, and it also follows that inputs $x_i$ with $\text{msb}(x_i) = \bot$ are all routed into the suffix. As a result, the output after the ranking-based routing is a sorted ternary sequence $0^i \bot^{j,k}$. Note further that the relative position of inputs $x_i$ with $\text{msb}(x_i) \in \{0, \bot\}$ are maintained while others may be permuted.

As the parity-based configurations also use $\rho_{\bot}(x_{j,q+\log(n)})$, we conclude that their configuration is again the same. □
Theorem 3 (Routing Partial Permutations): The RBS network with a front-end concentrator as constructed according to Figures 4 and 5 will correctly route any partial permutation if the circuits described in Sections III-B, III-C, and III-D are used as Split modules.

Proof: Finally, a little modification has to be made to the radix-based sorting network so that it can be used as an interconnection network for partial permutations: We therefore implement a TRBS network for \( n \) inputs \( x_0, \ldots, x_{n-1} \) as shown in Figure 5. Due to Figure 4, we can assume that the input is a prefix sequence \( b' \perp b' \) with boolean values \( b \).

Thus, using one of the three Split modules of the previous section, it will be sorted into a sequence \( u_0, \ldots, u_{n-1} \), where sorted means that \( \text{msb}(u_i) \leq \text{msb}(u_{i+1}) \) with ordering \( 0 \leq \perp \leq 1 \). Since the inputs \( x_0, \ldots, x_{n-1} \) must be a partial permutation of the addresses \( 0, \ldots, n-1 \), there are at most \( \frac{n}{2} \) inputs \( x_i \) with \( \text{msb}(x_i) = 0 \) and there are also at most \( \frac{n}{2} \) inputs \( x_i \) with \( \text{msb}(x_i) = 1 \). Hence, we conclude that the halves \( u_0, \ldots, u_{\frac{n}{2}-1} \) and \( u_{\frac{n}{2}}, \ldots, u_{n-1} \) must be of the form \( 0^m \perp 2^{-m} \) and \( 1^m - 1^k \), respectively. For this reason, we reverse the upper half \( \perp 2^{-k} - 1^k \) by the Mirror module shown in Figure 5 so that the inputs to the two TRBS modules become prefix sequences \( u_0, \ldots, u_{\frac{n}{2}-1} \) and \( u_{\frac{n}{2}}, \ldots, u_{n-1} \). Since both halves are now again prefix sequences, it follows by the induction hypothesis that these are also routed correctly by a TRBS module. □

Hence, the circuits we constructed using the RB-FS-RV network are able to sort all ternary prefix sequences by little modifications: We just have to consider \( \varrho_L(x_{i,q} + \log(n)) := x_{i,q} \land x_{i,q} + \log(n) \) for computing the ranks or the parities to achieve this. Since arbitrary ternary sequences are converted to ternary prefix sequences by the front-end concentrator, and since our Split modules also generate again prefix sequences, these RBS networks can handle all partial permutations.

B. Constructing Split Modules as Ternary Sorters

In the previous section, we showed how the RBS network built by Split modules as discussed in Section III can route all partial permutations. This is not possible for arbitrary RBS networks using other Split modules. For this reason, we consider in this and the following section (see also [16]) how to generate ternary sorters from binary sorters. Using the ternary sorters as Split modules leads to other RBS networks that can handle all partial permutations.

The left-hand side of Figure 6 shows how a ternary sorter can be constructed by two binary sorters that we call the 0-sorter and the 1-sorter, respectively. Both binary sorters obtain the \( n \) inputs \( x_0, \ldots, x_{n-1} \) after a pre-processing step that modifies the msbs \( x_{i,q} + \log(n) \) of the invalid target addresses as shown on the upper right part of Figure 6 as \( x_{i,q}^0 + \log(n) \) and \( x_{i,q}^1 + \log(n) \) for the 0- and 1-sorter, respectively. Note that after the pre-processing step, only the valid inputs have msbs 0 and 1 for the 0- and 1-sorter, respectively.

After this, the 0-sorter and the 1-sorter sort their input sequences to output sequences \( l_0, \ldots, l_{n-1} \) and \( u_0, \ldots, u_{n-1} \), respectively, by only considering the modified msbs \( x_{i,q}^0 + \log(n) \) and \( x_{i,q}^1 + \log(n) \). Hence, the 0-sorter uses the ordering \( 0 \prec \perp \prec 1 \) while the 1-sorter uses ordering \( 0 \prec \perp \prec 1 \) (regarding the original inputs).

The lower right part of Figure 6 shows how the 0- and 1-sorter’s output sequences look like in general: The 0-sorter’s output sequence starts with values \( (l_{i,q}, l_{i,q} + \log(n)) = (1, 0), \) i.e., \( 0 \), followed by values \( (l_{i,q}, l_{i,q} + \log(n)) = (1, 1), \) i.e., \( 1 \), while the 1-sorters output sequence starts with values \( (u_{i,q}, u_{i,q} + \log(n)) = (0, 0), \) i.e., \( 0 \), followed by values \( (u_{i,q}, u_{i,q} + \log(n)) = (1, 1), \) i.e., \( 1 \). The final stage of multiplexers will then determine output \( y_i \) by selecting one of the corresponding values \( l_i \) or \( u_i \) as follows where \( l_i \) is obtained from \( l_i \) by setting its valid bit to 0:

\[
y_i := \begin{cases} 0 & \text{if } l_{i,q} \land l_{i,q} + \log(n) \\ l_i & \text{if } l_{i,q} \land \neg l_{i,q} + \log(n) \\ u_i & \text{otherwise} \end{cases}
\]

Note that the number of valid inputs can be at most \( n \) hence, we never have both \( u_{i,q} \land l_{i,q} + \log(n) \) and \( l_{i,q} \land \neg l_{i,q} + \log(n) \). Note further that we have to set \( l_i := 0 \) in case \( l_i \) is chosen for \( y_i \), but \( l_{i,q} \land \neg l_{i,q} + \log(n) \) does not hold (this way, we avoid that an input with \( (x_{i,q}, x_{i,q} + \log(n)) = (1, 1) \) will be taken from the 0-sorter that has already been copied from the 1-sorter). It can be easily verified that the circuit shown in Figure 6 implements a ternary sorter, i.e., any input sequence \( x_0, \ldots, x_{n-1} \) of values \( \{0, \perp, 1\} \) is correctly sorted using the total order \( 0 \prec \perp \prec 1 \).

C. Constructing Split Modules as Ternary Concentrators

Split modules do not have to be ternary sorters to partition the inputs according to their msbs. Instead, it is sufficient to route all \( x_i \) with \( \text{msb}(x_i) = \perp \) to the upper half and those with \( \text{msb}(x_i) = 0 \) to the lower half, while those with \( \text{msb}(x_i) = 1 \) may be routed to any half.

For this reason, we can also consider the slightly simplified construction given in Figure 7. Compared to Figure 6, we modify the msbs \( x_{i,q} + \log(n) \) of the target addresses in the same way, but additionally invalidate all 1s and 0s in the 0- and 1-sorter, respectively, as shown in the upper right part of Figure 7. Hence, the 0-sorter will only have inputs \( (x_{i,q}^0, x_{i,q}^0) \in \{0, 1\}, \) i.e., \( \perp, 0 \) and the 1-sorter will only have inputs \( (x_{i,q}^1, x_{i,q}^1) \in \{0, 1\}, \) i.e., \( \perp, 1 \). We can simply determine outputs \( y_i \) as shown in Figure 7, i.e., we take the lower half from the 0-sorter and the upper half from the 1-sorter. While not yielding a ternary sorter for general ternary sequences, Figure 7 still sorts all ternary input sequences that will appear in RBS networks for partial permutations (i.e., sequences where at most \( \frac{n}{2} \) inputs \( x_i \) are 0 and 1).
Split(n) := Sort(n/2) \perp \text{Sort}(n/2) \perp \text{HC}(n)

D. Constructing Split Modules by Sorters and Half Cleaners

In previous work [13], [15], we have shown how Split modules with \( n \) inputs/outputs can be constructed as shown in Figure 8 using two (ternary) sorters with \( n/2 \) inputs/outputs and a half cleaner circuit. Half cleaners were introduced by Batcher in [1] for the construction of his bitonic sorting networks. We observed that half cleaners can also be used to implement binary [13] and ternary [15] Split modules as shown in Figure 8. Due to lack of space, we cannot list details of the definition of half cleaners, and just mention here that these circuits have size \( O(n) \) and depth \( O(1) \), so that the depth is mainly determined by the used sorting networks (see [13], [15] for further details). Using Split modules built by half cleaners and sorters of half the size instead of just the sorters significantly improves the circuit size and depth.

As outlined in [15], it is required to use sorting networks for the construction of Figure 8. In particular, the construction shown in Section IV-C and Figure 7 cannot be used. Moreover, the half-cleaner construction does not maintain prefix sequences, so that it cannot be used for the construction of Section IV-A either. Hence, we cannot combine the half-cleaner optimization with the already optimized networks in Sections IV-A and IV-C.

V. Experimental Results

We have implemented the three RBS networks of Sections III-B, III-C, and III-D with the extension to route partial permutations of Section IV-A. For the generated circuit netlists, we counted the number of gates and also the depth of the circuits as shown in Table III. As can be seen, the depth of the original circuit due to [27] (first part of the table) is not acceptable since it grows linearly with the number of inputs \( n \). The depth of the parallel computation of the same circuit grows with \( O(\log(n)^2) \) while the depth of the ranking-based configuration only grows with \( O(\log(n)) \). However, that difference is not significant up to 1024 components and leads to a comparable depth of both. Moreover, the size of the first two networks is almost the same, but worse for the third. We therefore conclude that the best network is the one obtained by the parallel computation of the parity bits.
VI. Summary

We presented three ways to configure a particular permutation network (RB-FS-RV) as a binary sorter. As such, it can be used as a Split module in radix-based sorting and interconnection networks. We proved that all three configurations maintain prefix sequences, and that we can therefore extend them to route all partial permutations by a single front-end concentrator. Hence, all three considered configurations can be used for partial permutations as well, and therefore lead to very efficient implementations of nonblocking interconnection networks.

REFERENCES