How to Sort N Items Using a Sorting Network of Fixed I/O Size

Stephan Olariu, *Member*, *IEEE Computer Society*,
M. Cristina Pinotti, *Member*, *IEEE Computer Society*, and
S.Q. Zheng, *Senior Member*, *IEEE*

Abstract—Sorting networks of fixed I/O size p have been used, thus far, for sorting a set of p elements. Somewhat surprisingly, the important problem of using such a sorting network for sorting arbitrarily large data sets has not been addressed in the literature. Our main contribution is to propose a simple sorting architecture whose main feature is the pipelined use of a sorting network of *fixed* I/O size p to sort an arbitrarily large data set of N elements. A noteworthy feature of our design is that no extra data memory space is required, other than what is used for storing the input. As it turns out, our architecture is feasible for VLSI implementation and its time performance is virtually independent of the cost and depth of the underlying sorting network. Specifically, we show that by using our design N elements can be sorted in $\Theta(\frac{N}{p}\log\frac{N}{p})$ time without memory access conflicts. Finally, we show how to use an AT^2-optimal sorting network of fixed I/O size p to construct a similar architecture that sorts N elements in $\Theta(\frac{N}{p}\log\frac{N}{p\log p})$ time.

Index Terms—Computer architecture, sorting, parallel processing, pipelined processing, sorting networks.

1 Introduction

Sorting networks are a well-studied class of parallel sorting devices. For an early treatment of the subject, see [5], [14]; for recent surveys, we refer the reader to [2], [3], [8], [22], [25], [26]. In general, sorting networks are suitable for VLSI realization. This is due, in part, to the fact that the processing elements are typically simple comparators and the structure of the network is fairly regular.

Several parameters are used to characterize the quality of a sorting network $\mathcal{T}$. The *cost* of $\mathcal{T}$, denoted by $C(\mathcal{T})$, is the number of constant fan-in processing nodes in the network. The *depth* of $\mathcal{T}$, denoted by $D(\mathcal{T})$, is the maximum number of nodes on a path from an input to an output. For example, Batcher's classic bitonic sorting network and odd-even merge sorting network [5], [6] have cost $O(p\log^2 p)$ and depth $O(\log^2 p)$, where p is the network I/O size. The *time performance* of a sorting network is the number of parallel steps performed, which is the depth of the network. Ajtai et al. [1] proposed a sorting network, commonly called the AKS sorting network, of I/O size p, depth $O(\log p)$, and cost $O(p\log p)$. Later, Leighton [16] and Paterson [24] developed comparator-based sorting networks of I/O size p and cost $O(p)$ that sort p elements in $O(\log p)$ time. The AKS network is both cost-optimal and depth-optimal (i.e., time-optimal) in the context of sorting p elements with each comparator used once. Leighton's network is cost-optimal and time-optimal in the context of sorting p elements with each comparator used more than once.

The goodness of a sorting network implemented in VLSI is assessed by combining the VLSI layout area and the time performance of the network. The most commonly used metric is the AT^2 complexity [27], where A is the chip area and T is the computation time. Several AT^2-optimal sorting networks under different word length models have been proposed in the literature [9], [11], [16], [24].

It is interesting to note that in spite of the fact that sorting networks of fixed I/O size p have been extensively investigated in the context of sorting p elements, their efficient use for sorting a large number, say N, of elements has not received much attention in the literature. In real-life applications, the number N of input elements to be sorted is much larger than p. In such a situation, the sorting network must be used repeatedly, in a pipelined fashion, in order to sort the input elements efficiently. Assume that the input as well as the partial results reside in several constant-port memory modules. Then, scheduling memory accesses and the I/O of the sorting network becomes the key to achieving the best possible sorting performance. Clearly, if an appropriate answer to this problem is not found, the power of the sorting network will not be fully utilized.

The problem of building sorting networks out of smaller components, such as sorters and mergers, has received attention in the literature [8], [9], [12], [18], [23], [28]. For example, Bilardi and Preparata [9] use a tree of mergers of various sizes. Drysdale and Young [12] construct a k-way merge sorting network using 2-sorters as the basic building blocks. Nassimi and Sahni [18] construct sorting networks by using various combinations of mergers. Tseng and Lee [28] construct a sorting network of I/O size p^2 using $O(p)$ layers of p-sorters. Recently, Parker and Parberry [23] showed that for arbitrary N, a sorting network of I/O size

- *S. Olariu is with the Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162. E-mail: olariu@cs.odu.edu.*
- *M.C. Pinotti is with the Istituto di Elaborazione dell'Informazione, C.N.R, Pisa 56126, Italy.*
- *S.Q. Zheng is with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688.*

Manuscript received 23 Dec. 1997; revised 17 Aug. 1998.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 106090.

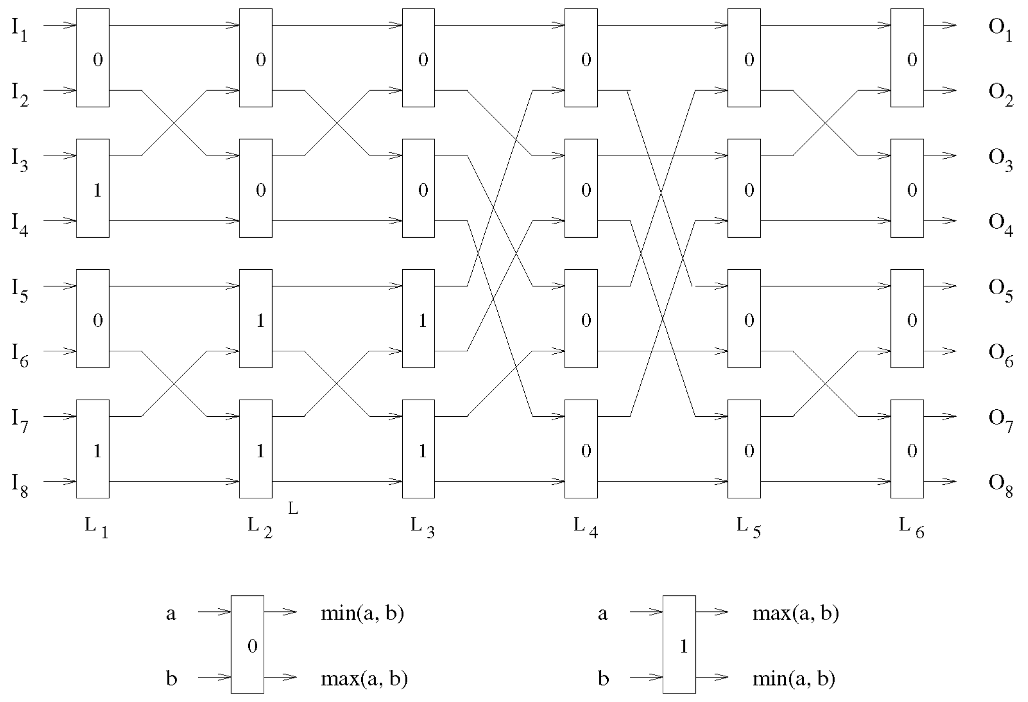


Fig. 1. A bitonic sorting network of I/O size 8.

N can be constructed using p-sorters, where p is a power of two, thus answering an open question posed in [14].

The related problem of sorting N elements by repeatedly using a p-sorter has received some attention as well [7], [21], [23]. A p-sorter is a sorting device capable of sorting p elements in constant time. Computing models for a p-sorter do exist. For example, it is known that p elements can be sorted in $O(1)$ time on a reconfigurable mesh of size $p \times p$ [13], [17], [19], [20]. A reconfigurable mesh is a multiprocessor system in which the processors are connected by a bus system whose configuration can be dynamically changed to suit computational needs. (We note here, for the sake of completeness, that Atallah et al. [4] have investigated the case of a p-sorter that takes p time to sort p items.)

Beigel and Gill [7] showed that the task of sorting N elements, $N \geq p$, requires $\Omega\left(\frac{N \log N}{p \log p}\right)$ calls to a p-sorter. They also presented an algorithm to sort N elements using $\Theta\left(\frac{N \log N}{p \log p}\right)$ calls to the p-sorter. Their algorithm, however, assumes that the p elements to be sorted by the p-sorter can be fetched in unit time, regardless of their location in memory. Since, in general, the address patterns of the operands of p-sorter operations are irregular, it appears that the algorithm of [7] cannot realistically achieve the time complexity of $O\left(\frac{N \log N}{p \log p}\right)$, unless one can solve in constant time the addressing problem on realistic machines. To address this problem, Olariu and Zheng [21] proposed a p-sorter-based architecture that sorts N elements in $O\left(\frac{N \log N}{p \log p}\right)$ time while strictly enforcing conflict-free memory accesses. In conjunction with the results of [7], their result completely resolves the time complexity issue of sorting N elements using a p-sorter. As it turns out, however, a p-sorter is a much more expensive device than a sorting network and its use should be avoided whenever possible. Besides, it is not clear whether it is possible to replace the p-sorter with a pipelined sorting network in the architecture of [21], while guaranteeing the same time performance. After this paper had been submitted, we found out about a similar effort by Lee and Batcher [15]. As it turns out, [15] presents two pipelined sorting algorithms, the faster of which runs in $O(\frac{N}{p}(\log \frac{N}{p} + \log^2 p))$ time.

The main contribution of this work is to propose a simple sorting architecture whose main feature is the pipelined use of a sorting network of fixed I/O size p to sort an arbitrarily large number N of elements. Specifically, we show that by using our design, N elements can be sorted in $\Theta(\frac{N}{p} \log \frac{N}{p})$ time. Clearly, our results are better than those presented in [15].

Our design consists of a sorting network of fixed I/O size p, a set of $\frac{p}{2}$ random-access memory modules, and a control unit. The memory access patterns are regular: In one step, elements from two rows of memory modules are supplied as input to the sorting network and/or the output of the sorting network is written back into two memory rows. Our architecture is feasible for VLSI implementation.

We take this idea one step further and show how to use an AT^2-optimal sorting network of fixed I/O size p to construct a similar architecture that sorts an arbitrary number N of elements in $\Theta(\frac{N}{p} \log \frac{N}{p \log p})$ time. An important feature of both architectures is that no extra data memory space is required, other than what is needed for storing the input.

The remainder of the paper is organized as follows: In Section 2, we discuss the details of the proposed architecture. In Section 3, we show how to obtain row-merge schedules, a critical ingredient for the efficiency of our design. Section 4 extends the results of Sections 2 and 3 by showing how to use an AT^2-optimal sorting network of fixed I/O size p to obtain an architecture to sort N elements in $\Theta(\frac{N}{p}\log\frac{N}{p\log p})$ time. Section 5 offers concluding remarks and poses a number of open problems. Finally, the Appendix contains a detailed example of how our design works.

2 The Architecture

A sorting network can be modeled by a directed graph whose nodes represent processing elements and whose edges represent the links connecting the nodes, as illustrated in Fig. 1. The processing elements can be simple comparators or more complex processors capable of performing arithmetic operations. A *comparator* has two inputs and two outputs and is used to perform a compare-exchange operation. A *comparator-based sorting network* is a sorting network whose processing elements are comparators. In the remainder of this work, we use the term *sorting network* to refer exclusively to a comparator-based sorting network. Fig. 1 and Fig. 2 illustrate Batcher's classic sorting networks, with I/O size 8. As illustrated in Fig. 1, two types of comparators are used. For a type 0 comparator, the smaller and larger of the two input numbers emerge, after comparison, at the top and bottom output, respectively. A comparator of type 1 produces the output in reverse order. Unless stated otherwise, we assume that when a sorting network of fixed I/O size p is used to sort p elements, each of its comparators is used exactly once.

Referring to Figs. 1 and 2, we say that a sorting network $\mathcal{S}$ is *layered* if each of its comparators is assigned to one of the $D(\mathcal{S})$ layers L_k, $1 \le k \le D(\mathcal{S})$, defined as follows:

- Assign to layer L_1 the comparators whose inputs are outputs of no other comparators in the network and exclude them from further consideration;
- For every k, $2 \le k \le D(\mathcal{S})$, assign to layer L_k the comparators whose inputs are outputs of comparators in layers L_i and L_j, $1 \le i, j < k$ and exclude them from further consideration.

A simple inductive argument shows that for every k, $2 \le k \le D(\mathcal{S})$, every comparator in layer L_k receives at least one input from a comparator in layer L_{k-1}. Therefore, in a layered sorting network, all longest paths from the network input to the comparators in layer L_k must have the same length[1] k.

We say that a sorting network $\mathcal{S}$ is *pipelined* if, for every k, $2 \le k \le D(\mathcal{S})$, all paths from the network input to the comparators in layer L_k have the same length. As an illustration, the bitonic sorting network shown in Fig. 1 is a pipelined network, whereas the odd-even merge sorting network of Fig. 2 is not. The intuition for this terminology is that a pipelined sorting network $\mathcal{S}$ of I/O size p can be used to sort sets of p elements concurrently in a pipelined fashion. It is easy to confirm that in a pipelined network $\mathcal{S}$ each layer contains exactly $\frac{p}{2}$ comparators.[2]

1. The length of a path is taken to be the number of edges on the path.
2. For convenience, we assume that p is even. The case where p is odd is similar.

Given a sorting network $\mathcal{S}$, one can always introduce additional buffer nodes (latches), if necessary, in such a way that the nodes of the resulting network—comparators and latches—can be partitioned into stages $S_1, S_2, \cdots, S_{D(\mathcal{S})}$. Specifically, for every k, $1 \le k \le D(\mathcal{S})$, place all the comparators in layer L_k in S_k. If one of the two inputs of a comparator c in layer L_k is the output of a comparator c' in layer L_i with $i < k-1$, then add a sequence of $k-i-1$ latch nodes $l'_1, l'_2, \cdots, l'_{k-i-1}$ on the edge from c' to c so that l'_j is in stage S_{i+j}. Likewise, for each output of a comparator c in layer L_k such that $k < D(\mathcal{S})$ that is also the output of the network, we add $D(\mathcal{S}) - k$ latch nodes $l'_{k+1}, l'_{k+2}, \cdots, l'_{D(\mathcal{S})-k}$ on the output edge. The reader should have no difficulty confirming that in the resulting network all paths from the network input to the nodes in the same layer have the same length. Thus, this transformation converts a nonpipelined network into a pipelined one. For example, after adding latches to the odd-even merge sorting network of Fig. 2, we obtain the network shown in Fig. 3.

Our proposed architecture, that we call the *Row-Merge Architecture* (RMA, for short), is illustrated in Fig. 4 for $p = 8$. The RMA consists of the following components:

1. A pipelined sorting network $\mathcal{T}$ of fixed I/O size p with inputs $I_1, I_2, \ldots, I_p$ and with outputs $O_1, O_2, \ldots, O_p$.
2. $\frac{p}{2}$ constant-port memory modules $M_1, M_2, \ldots, M_{\frac{p}{2}}$ collectively referred to as the *data memory*. For every k, $1 \le k \le \frac{p}{2}$, memory module M_k is connected to inputs I_k and $I_{\frac{p}{2}+k}$ and to outputs O_k and $O_{\frac{p}{2}+k}$ of the sorting network.
3. A *control unit* (CU) consisting of a *control processor* (CP) and of a *control memory*.

The words with the same local address in all memory modules are collectively referred to as a *memory row*. The N input elements are stored, as evenly as possible, in $\frac{2N}{p}$ consecutive memory rows. Dummy elements of value $+\infty$ are added, as necessary, to ensure that all memory modules contain the same number of elements. These dummy elements will be removed after sorting. The read/write operations are carried out in a single instruction (address) stream multiple data stream fashion controlled by the CU. Specifically, the CU is responsible for generating memory access addresses: In every step, the same address is broadcast to all memory modules which use it as the local address for the current read or write operation. We assume that the address broadcast operation takes constant time. The CU can disable memory read/write operations, as necessary, by appropriately using a mask.

When operating in pipelined fashion, in a generic step i, p elements from two memory rows are fed into the sorting network. At the end of step $i + D(\mathcal{T}) + 1$, the sorted sequence of p elements from these two rows emerges at the output ports of $\mathcal{T}$ and is written back into two memory rows. This process is continued until all the input elements have been sorted. To simplify our analysis, we assume that one memory cycle is sufficient for reading, writing, and for comparator operations. This assumption is reasonable if each memory module has two ports for reading and two ports for writing. If each memory module has only one port

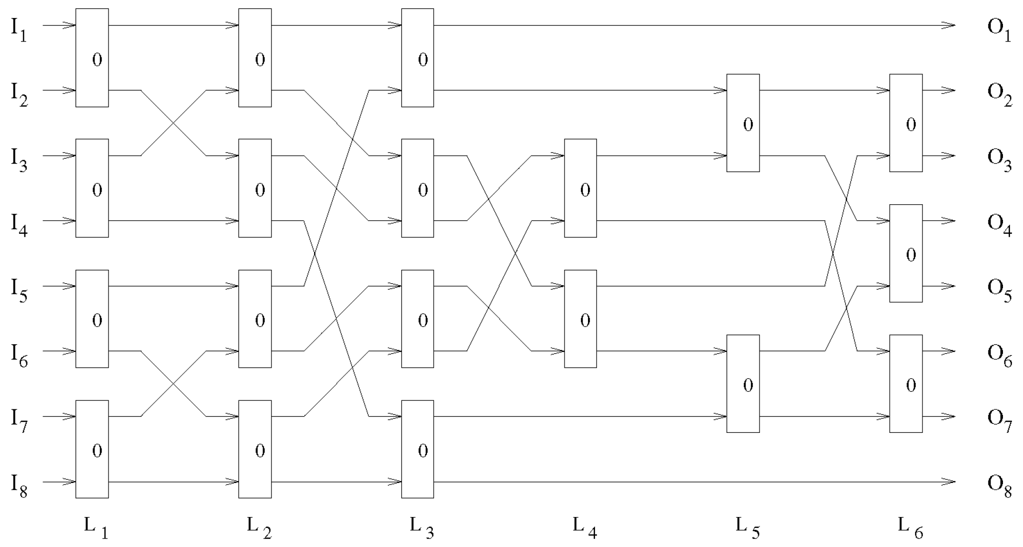


Fig. 2. An odd-even merge sorting network of I/O size 8.

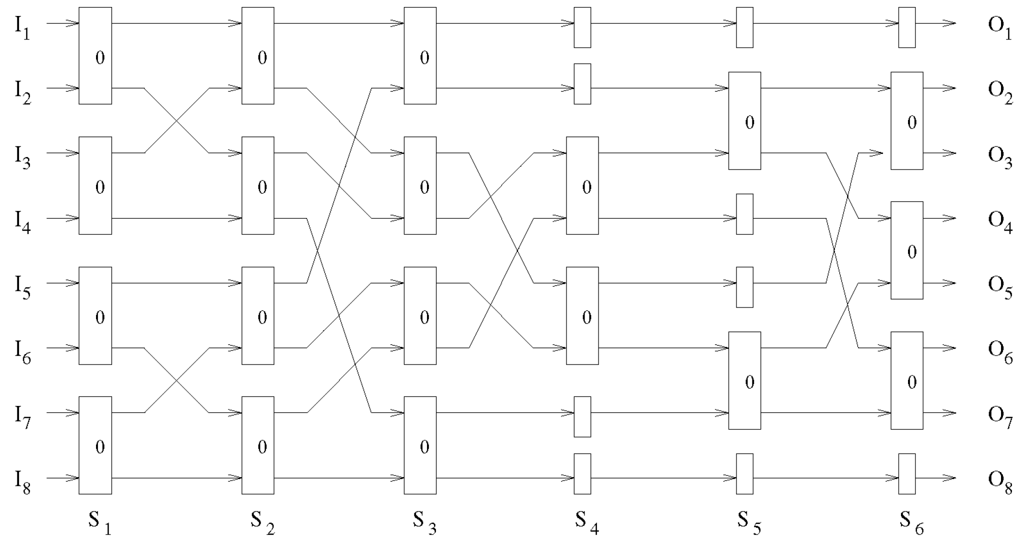


Fig. 3. The odd-even merge sorting network with latches added.

for both reading and writing, the performance degrades by a small constant factor.

Let (a, b) be an ordered pair of memory rows in data memory. In the process of sorting, the elements in memory row a (resp. b) are read into the left (resp. right) half of the network input and the corresponding elements are sorted in nondecreasing order. Finally, the left (resp. right) half of the resulting sorted sequence emerging at the network output is written back into data memory to replace the original row a (resp. b). It is now clear why we refer to our design as the *Row-Merge Architecture*.

In order to sort efficiently the $\frac{2N}{p}$ memory rows of the RMA, we wish to identify a finite sequence $MS = <(a_1, b_1), (a_2, b_2), \cdots, (a_s, b_s)>$ of pairs of memory rows such that, by following this sequence, the elements are sorted in row-major order. We call such a sequence MS a *row-merge schedule* or, simply, a *merge schedule*.

At this time, the reader may wonder about the power of the RMA. In Theorem 1, we provide a partial answer to this question by establishing a lower bound on the time required by any algorithm that sorts N elements using the RMA.

Theorem 1. *Any algorithm that correctly sorts N elements on the RMA using a sorting network of I/O size p must take* $\Omega(\frac{N}{p}\log\frac{N}{p})$ *time.*

Proof. We ignore the time delay caused by the depth of the sorting network and, consequently, we assume that the sorting network takes $O(1)$ time to sort any group of p elements. This assumption is reasonable, since it can only help the algorithm. We show that even with this favorable assumption, any sorting algorithm using the RMA must take $\Omega(\frac{N}{p}\log\frac{N}{p})$ time.

For this purpose, we only need to exhibit a particular sequence of $\frac{2N}{p}$ memory rows for which any algorithm operating in the RMA requires $\Omega(\frac{N}{p}\log\frac{N}{p})$ time. Consider an arbitrary sequence $a_1, a_2, \ldots, a_{\frac{2N}{p}}$ of real numbers stored in an array $A[1..\frac{2N}{p}]$ such that for every i,

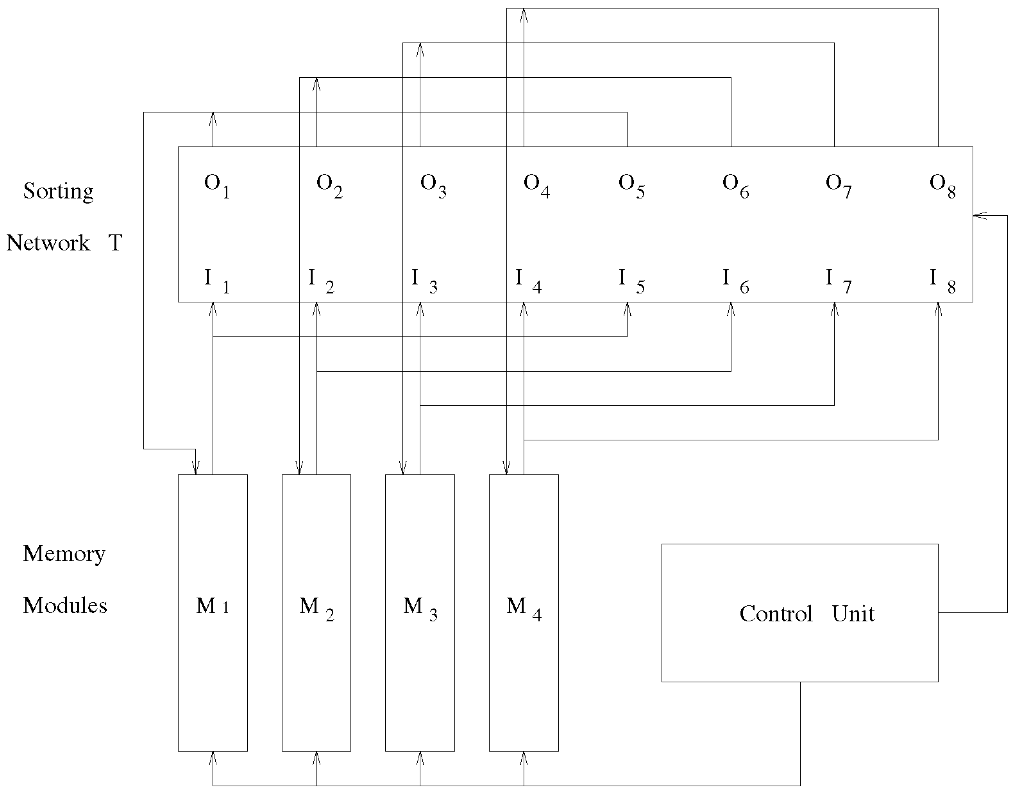


Fig. 4. The Row-Merge Architecture with $p = 8$.

$1 \leq i \leq \frac{2N}{p}$, $A(i) = a_i$. The $\frac{2N}{p}$ memory rows are assumed to be such that for every i, $1 \leq i \leq \frac{2N}{p}$, all the $\frac{p}{2}$ words in memory row i are equal to a_i.

Now, consider an arbitrary merge schedule MS that correctly sorts the resulting $\frac{2N}{p}$ memory rows. From this MS we construct an algorithm B, in the comparison tree model, that sorts the array A. The idea is that algorithm B simulates on A the actions of MS on the set of memory rows. More precisely, when MS sorts memory rows i and j, algorithm B compares and, if necessary, interchanges the entries $A(i)$ and $A(j)$.

Now assume that the merge schedule MS sorts the $\frac{2N}{p}$ memory rows in $o(\frac{N}{p}\log\frac{N}{p})$ time. The simulation just described implies that algorithm B sorts the array A in $o(\frac{N}{p}\log\frac{N}{p})$ time. However, this is impossible as the lower bound for sorting $\frac{2N}{p}$ elements in the comparison tree model is $\Omega(\frac{N}{p}\log\frac{N}{p})$. This completes the proof of the theorem. □

3 Generating Row-Merge Schedules

In order to sort N input elements correctly and efficiently on the RMA, we need to find a merge schedule $MS =< (a_1, b_1), (a_2, b_2), \cdots, (a_s, b_s) >$ to guide the computation. The MS specifies, in left to right order, the pairs of memory rows that will be supplied as input to the sorting network, in a pipelined fashion. Thus, the ordered pair (a_1, b_1) is supplied in the first time unit, followed by the ordered pair (a_2, b_2) in the second time unit, and so on. For reasons that will be discussed shortly, we are interested in merge schedules that satisfy the following two conditions necessary to ensure the correctness of our scheme:

(1) The RMA must sort correctly, even if a p-sorter is used instead of the sorting network, that is, for sorting networks of depth one.
(2) No row number should appear more than once in any subsequence

$$(a_i, b_i), (a_{i+1}, b_{i+1}), \cdots, (a_{i+D(\mathcal{T})-1}, b_{i+D(\mathcal{T})-1})$$

of length $D(\mathcal{T})$ of MS.

In addition, our goal is to produce a merge schedule of length $O(\frac{N}{p}\log\frac{N}{p})$ to match the lower bound of Theorem 1.

A p-sorter can be perceived as a sorting network $\mathcal{T}$ of I/O size p and depth $D(\mathcal{T}) = 1$. Therefore, condition (1) is necessary for correctly sorting N elements in general. For $D(\mathcal{T}) > 1$, if condition (2) is violated, the MS may not guide the pipelined operations to correctly sort the N elements because of possible data dependencies. To see this, consider the pairs of memory rows (a_i, b_i) and (a_{i+j}, b_{i+j}) such that $j < D(\mathcal{T})$ and $a_i = a_{i+j}$. Then, in steps $i + D(\mathcal{T}) + 1$ and $i + j + D(\mathcal{T}) + 1$, memory rows a_i, b_i and b_{i+j} are updated. It is possible that an element that is originally in row a_i is duplicated in both rows b_i and row b_{i+j}, while an element that is originally in row b_i or row b_{i+j} is lost.

The remainder of this section will present a detailed discussion of the "network-to-MS" conversion process, that is, the process of generating a suitable row-merge schedule from a given sorting network $\mathcal{S}$. In outline, we construct an *augmented* network $\mathcal{S}''$ of S that satisfies a number of properties that we also discuss.

The central idea of our approach is motivated by the following well-known fact mentioned in [10].

Proposition 1. *For any parallel algorithm that uses compare-exchange operations to sort m elements with m processors, there is a corresponding algorithm to sort rm elements with m processors, where every comparison in the first algorithm is replaced by a merge-sorting of two lists of r elements in the second.*

For later reference, we refer to Proposition 1 as the *compare-exchange/merge-split principle*.

Let $\mathcal{S}$ be a sorting network of I/O size $m = \frac{2N}{p}$. Knuth [14] suggested representing $\mathcal{S}$ in the way shown in Fig. 5b. Specifically, there are m horizontal lines, each labeled by an integer i. The left and right endpoints of the line labeled i represent, respectively, the ith input and the ith output of the network. A comparator, represented by a directed vertical segment originating at line i and ending at line j, causes an interchange of its inputs, if necessary, so that the smaller number appears on line i and the larger number appears on line j. We call this representation the *line representation* of $\mathcal{S}$.

Constructing the line representation of a sorting network $\mathcal{S}$ of I/O size m from the graph representation of $\mathcal{S}$ is straightforward and can be found in [14]. Nonetheless, for the sake of completeness, we now briefly illustrate the construction. We perceive a comparator as a 2×2 switch of two states, straight and cross. We assign an input value i to the network input I_i, $1 \leq i \leq m$, and set all switches (comparators) to the straight state. Then, the m input values propagate through the switches. The values j and k received by a switch define the corresponding vertical segment with endpoints on lines j and k in the line representation of $\mathcal{S}$. The type of the corresponding comparator determines the direction of the vertical segment. Fig. 6 illustrates this propagation process for the network of Fig. 1. A pair of integers and an arrow at the input of each comparator define the directed vertical line segment in the line representation shown in Fig. 5b.

Assume that the N elements to sort are located in memory rows 1 through m. We generate a merge schedule MS from the line representation and the layer partition of $\mathcal{S}$ by the following greedy algorithm. Initially, the inputs to all comparators are *unmarked*. Let C_1 be an arbitrary FIFO (first-in-first-out) queue of comparators at level L_1. We obtain the FIFO queue C_{i+1} of comparators in level L_{i+1} as follows: Set C_{i+1} to empty. Scan the comparator queue C_i in order and, for each comparator in C_i, mark its two output edges. As soon as the two input edges of a comparator c are marked, enqueue c into C_{i+1}. At this point, the reader will not fail to note that comparator c must, indeed, belong to layer L_{i+1}. This process is continued, as described, until all queues C_j, $1 \leq j \leq D(\mathcal{S})$, have been constructed. Finally, the queues C_j are concatenated in order to obtain a sequence C of comparators such that C_i precedes C_{i+1}.

Let $C = (c_{k_1}, c_{k_2}, \cdots, c_{k_s})$, $s = C(\mathcal{S})$, be the sequence of comparators obtained from $\mathcal{S}$ using the greedy algorithm just described. With each comparator c_{k_j} whose corresponding directed vertical segment originates at line a_j and ends at line b_j in the line representation of $\mathcal{S}$, we associate the ordered pair (a_j, b_j). Let

$$MS =< (a_1, b_1), (a_2, b_2), \cdots, (a_s, b_s) >$$

be the resulting sequence of ordered pairs corresponding to C. Consider the correspondence between the data memory and the line representation of $\mathcal{S}$ such that the horizontal line i corresponds to memory row i. The compare-exchange/merge-split principle (Proposition 1) guarantees that the elements in memory can be sorted in row-major order if we merge the rows sequentially, following the merge schedule MS. Fig. 5 illustrates the correspondence between a data memory of four modules, eight words per module (i.e., $m = 8$), and the line representation of the network in Fig. 1. For example, by applying the greedy algorithm to the line representation in Fig. 5b we obtain the following merge schedule:

$$\begin{aligned} MS =<&(1,2),(4,3),(5,6),(8,7),(1,3),(2,4),(7,5),(8,6),\\ &(1,2),(3,4),(6,5),(8,7),(1,5),(2,6),(3,7),(4,8),\\ &(1,3),(2,4),(5,7),(6,8),(1,2),(3,4),(5,6),(7,8)>. \end{aligned} \tag{1}$$

In the bitonic sorting network of I/O size m, assuming m to be even, there are exactly $\frac{m}{2}$ comparators per layer. As will be shown shortly, if $N > 2pD(\mathcal{T})$, any MS generated by the greedy algorithm from the bitonic sorting network of I/O size $m = \frac{2N}{p}$ satisfies conditions (1) and (2) above and, therefore, can be used to correctly sort N elements on the RMA. However, there exist sorting networks that cannot be used to generate a merge schedule that satisfies condition (2) for $D(\mathcal{T}) \geq 3$. This fact restricts the applicability of the MS generating scheme. For example, if $D(\mathcal{T}) \geq 3$, no MS generated directly from the network featured in Fig. 7 can satisfy condition (2). To remedy this problem, we introduce the concept of augmenting sorting networks.

Given an arbitrary sorting network $\mathcal{S}$ of I/O size m, the *augmented sorting network* $\mathcal{S}''$ of I/O size m derived from $\mathcal{S}$ is obtained as follows:

- Transform $\mathcal{S}$ into a pipelined network $\mathcal{S}'$ by adding latches as described in Section 2;
- Group the latches in each layer of $\mathcal{S}'$ into pairs (in an arbitrary way) and replace each pair of latches by "dummy" comparators;[3]
- If m is odd, delete the remaining latch in each stage.

The network obtained at the end of this simple algorithm is the desired augmented sorting network $\mathcal{S}''$. Clearly, in each layer of $\mathcal{S}''$ there are exactly $\lfloor \frac{m}{2} \rfloor$ comparators. We therefore assume that m is even. To be distinguished from real comparators, dummy comparators are represented by a node labeled d and by a vertical line segment without an arrow in the graph and line representation, respectively. For an illustration, the augmented network of the odd-even sorting network shown in Fig. 2 is given in Fig. 8.

3. Dummy comparators are introduced for convenience only. They amount to a "no-operation."

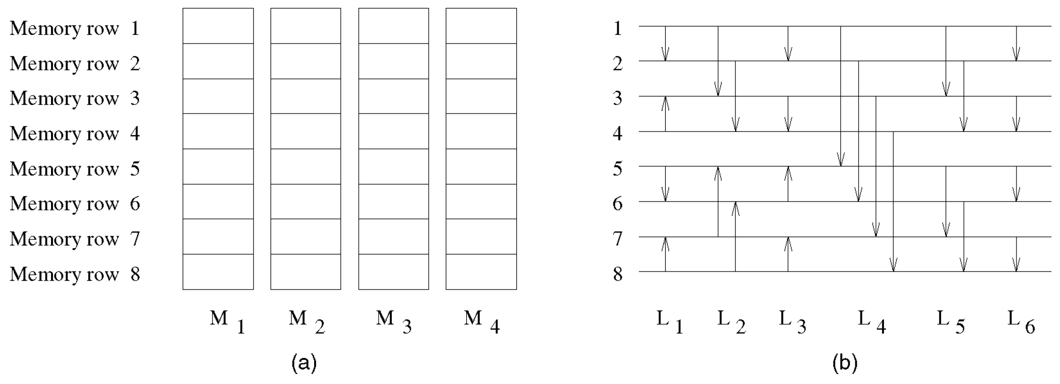


Fig. 5. Illustrating the correspondence between (a) memory address space and (b) line representation of a network $\mathcal{S}$. Here, $\mathcal{S}$ is the sorting network shown in Fig. 1.

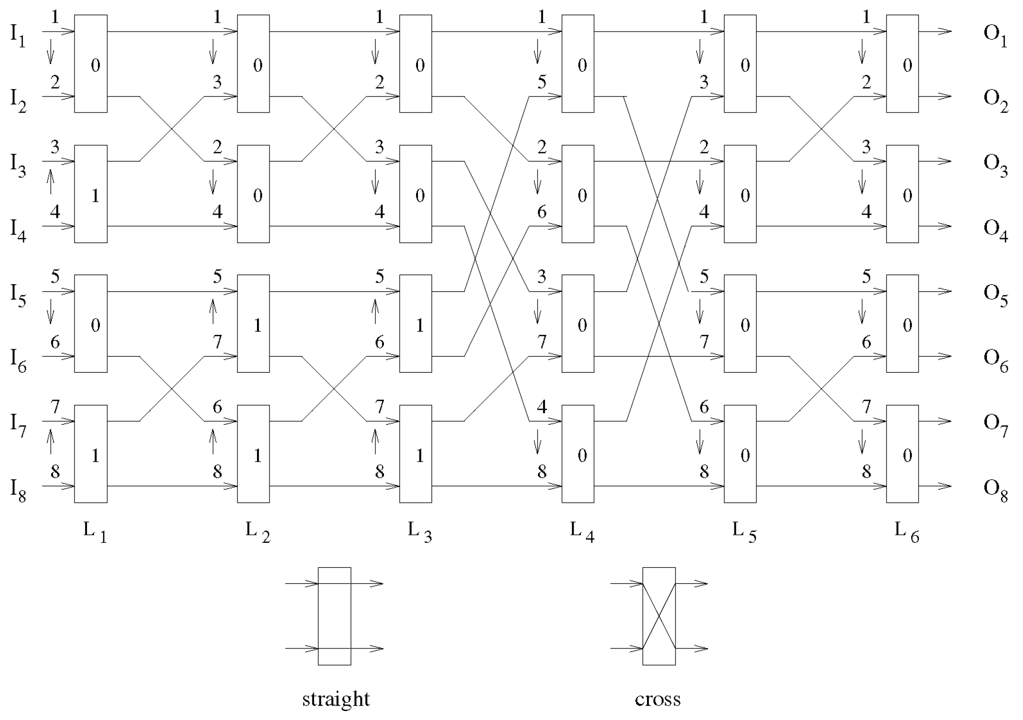


Fig. 6. Illustrating the method of obtaining the line representation of the network in Fig. 1.

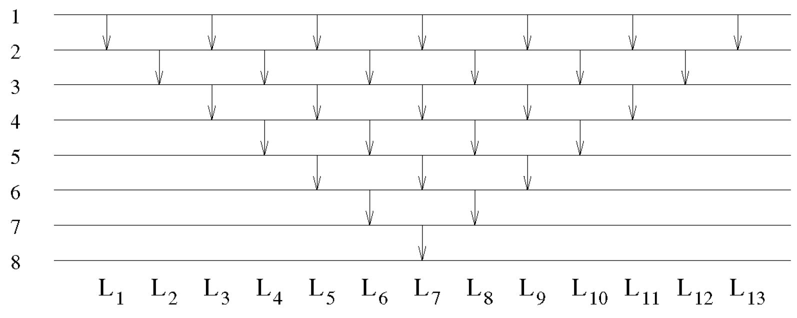


Fig. 7. Line representation of a variant of the bubble sort network.

We will still use our greedy algorithm to generate an MS from the augmented network $\mathcal{S}''$. The comparator selection process is exactly as described above. However, the task of translating a comparator sequence into the corresponding MS is slightly modified to accommodate dummy comparators. Specifically, when we translate the comparator sequence $C = (c_{k_1}, c_{k_2}, \cdots, c_{k_s})$ into the merge schedule

$$MS = < (a_1, b_1), (a_2, b_2), \cdots, (a_s, b_s) >,$$

if c_{k_j} is a dummy comparator then the order of the two row

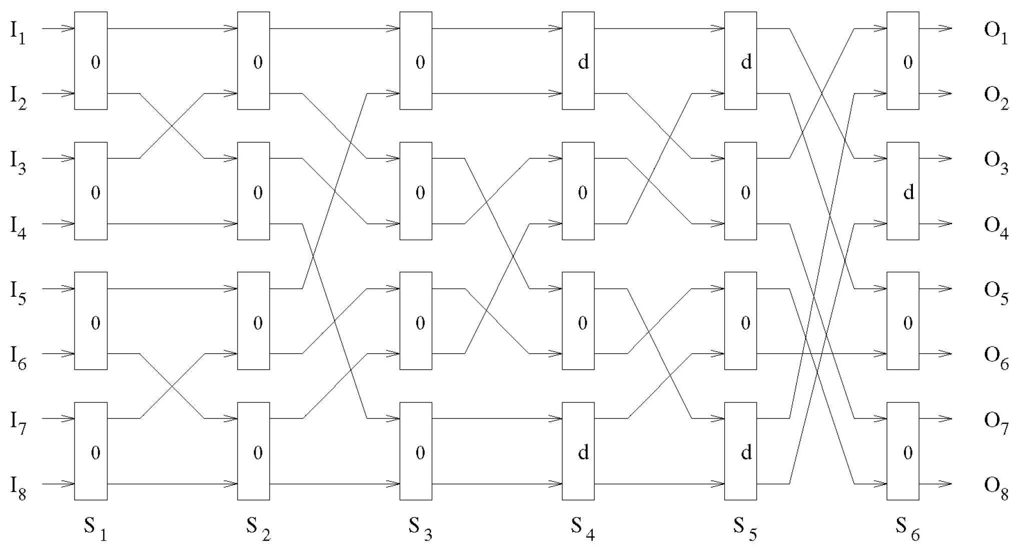


Fig. 8. The augmented odd-even merge sorting network.

numbers involved is arbitrary. By the same token, when we use the MS generated from an augmented network $\mathcal{S}''$, the write operation in step $i + D(\mathcal{T}) + 1$ is disabled if (a_i, b_i) corresponds to a dummy comparator of $\mathcal{S}''$. Note that this is possible since, as specified in the Section 2, our architecture supports masked write operations.

Operating in this fashion, the MS generated from an augmented network $\mathcal{S}''$ clearly satisfies condition (1) because of the compare-exchange/merge-split principle. As we shall prove in Theorem 2, the MS also satisfies condition (2). The length of the MS is the cost of the sorting network that was used to generate the MS. Both $\mathcal{S}$ and $\mathcal{S}''$ have the same depth, but $\mathcal{S}''$ has an increased cost compared with $\mathcal{S}$. We note that, at first, it would seem as though by using $\mathcal{S}''$ to derive a merge schedule, there will be a cost blowup in the transition from $\mathcal{S}$ to $\mathcal{S}''$. However, most of the sorting networks $\mathcal{S}$ of I/O size m, including the network featured in Fig. 7, have $O(mD(\mathcal{S}))$ cost and, therefore, the blowup is within a constant factor of the cost of $\mathcal{S}$. Moreover, the blowup is a constant for many of the networks in the literature.

To summarize our findings, we now state and prove the following important result:

Theorem 2. *Let $\mathcal{S}$ be an arbitrary sorting network of I/O size $2N/p$ and let $\mathcal{T}$ be a sorting network of I/O size p. If $N > 2pD(\mathcal{T})$, then any merge schedule obtained from the augmented network of $\mathcal{S}$ by the greedy algorithm can be used to correctly sort N elements in $O(\frac{N}{p}D(\mathcal{S}))$ time on the Row-Merge Architecture with $\mathcal{T}$ as the sorting device.*

Proof. We only need to show that if $N > 2pD(\mathcal{T})$, then any merge schedule generated from the augmented network $\mathcal{S}''$ of $\mathcal{S}$ satisfies condition (2).

For this purpose, let

$$MS =< (a_1, b_1), (a_2, b_2), \cdots, (a_s, b_s) >$$

be an arbitrary merge schedule corresponding to the comparator sequence $C = (c_{k_1}, c_{k_2}, \cdots, c_{k_s})$ generated from $\mathcal{S}''$ by the greedy algorithm. Conceptually, we treat the network $\mathcal{S}''$ as a data-driven (i.e., data-flow) architecture: The processing elements are precisely the comparators whose activation is driven by data availability. We say that a comparator c is ready for activation whenever its two inputs are available and it has not yet been used.

To prove the theorem, we need to show that for any j, such that all comparators preceding c_{k_j} have been used but c_{k_j} has not yet been used, all the comparators in the subsequence $(c_{k_j}, c_{k_{j+1}}, \cdots, c_{k_t})$ of C, where $t = \min\{j + \frac{N}{2p} - 1, s\}$, are ready for activation.

We let $|L_i| = \frac{N}{p}$ denote the number of comparators in layer L_i of $\mathcal{S}$; similarly, let $|C_i|$ stand for the length of the subsequence C_i constructed from the comparators in L_i by the greedy algorithm. Our construction guarantees that $|C_i| = |L_i| = \frac{N}{p}$. Initially, there are exactly $\frac{N}{p}$ comparators, all in C_1, ready for activation. Consider an arbitrary j, $j \leq s - \frac{N}{2p} + 1$, such that all comparators preceding c_{k_j} have been used and such that c_{k_j} is ready for activation. Suppose that c_{k_j} is the rth comparator in C_i. Then, the number of comparators in C_i that are ready for activation is, clearly, $\frac{N}{p} - r + 1$.

Now, if $r \leq \frac{N}{2p}$, then there are still at least $\frac{N}{2p} \geq D(\mathcal{T})$ comparators ready for activation in C_i and condition (2) holds in this case.

Similarly, if $r > \frac{N}{2p}$, then there are $\frac{N}{2p} - r + 1$ comparators ready for activation in C_i and at least $r - 1$ comparators that are ready for activation in C_{i+1} as a result of the activation of the first $r - 1$ comparators in C_i. Thus, in this case condition (2) holds as well.

If $j > s - \frac{N}{2p} + 1$, then all remaining comparators in C starting from c_{k_j} are ready for activation.

With this, we just proved that the merge schedule generated from the augmented network $\mathcal{S}''$ of $\mathcal{S}$ can be used as a merge schedule for network $\mathcal{T}$.

Note also that, by our previous discussion, the time required to sort N elements is $O(s)$, where s is the cost of $\mathcal{S}''$, and it is bounded above by $O(\frac{N}{p}D(\mathcal{S}))$. This completes the proof of the theorem. □

We note that Theorem 2 has the following important implications:

1. Any sorting network $\mathcal{T}$ whose depth $D(\mathcal{T})$ satisfies $N > 2pD(\mathcal{T})$ can be used to correctly sort N elements in pipelined fashion.
2. If $N \geq 2p^2$, any sorting network of depth no larger than p can be used as $\mathcal{T}$. Since the depth of virtually all practical sorting networks of I/O size p is smaller than p, any of these networks can be used as $\mathcal{T}$ in the RMA. It is important to note that this implies that the performance of the RMA is virtually *independent* of the sorting network $\mathcal{T}$ used as the sorting device.
3. If $N \geq 2p^2$, any row-merge schedule generated from the augmented network of any network $\mathcal{S}$ by our greedy algorithm can be used to sort N elements correctly, using as the sorting device a sorting network $\mathcal{T}$; in other words, the correctness of any merge schedule is independent of the sorting network $\mathcal{S}$ used to generate it.
4. The time required for sorting N elements is proportional to $D(\mathcal{S})$, the depth of $\mathcal{S}$.

We can select $\mathcal{T}$ from a wide range of sorting networks, depending on their VLSI feasibility. We also have a wide range of sorting networks from which to choose the network $\mathcal{S}$ for deriving merge schedules. We know that the depth of both the bitonic and the odd-even merge sorting network of I/O size m is $O(\log^2 m)$. Thus, using either of them as $\mathcal{S}$ to derive a merge schedule, N elements can be sorted in $O(\frac{N}{p}\log^2\frac{N}{p})$ time. It is well-known that the depth of the AKS sorting network of I/O size m is bounded by $O(\log m)$. Hence, using the AKS network as $\mathcal{S}$, N elements can be sorted in $O(\frac{N}{p}\log\frac{N}{p})$ time. Therefore, we state the following extension of Theorems 1 and 2.

Theorem 3. *The Row-Merge Architecture that uses a sorting network of I/O size p and depth at most p as the sorting device can sort N elements, $N \geq 2p^2$, in $\Theta(\frac{N}{p}\log\frac{N}{p})$ time.*

To the best of our knowledge, most of the known sorting networks, including the AKS network, are defined recursively. The graph representations of recursively defined sorting networks can be constructed in linear time. Given the graph representation of a sorting network $\mathcal{S}$ of I/O size $\frac{2N}{p}$, we now estimate the time it takes the CU to generate a merge schedule MS. The comparators in the first layer, L_1, of $\mathcal{S}$ can be easily identified. Next, the nodes of $\mathcal{S}$ can be divided into $D(\mathcal{S})$ layers as described in Section 2. Clearly, this process takes $O(C(\mathcal{S}))$ time. By scanning the nodes of $\mathcal{S}$ layer by layer, latches can be added to convert $\mathcal{S}$ into a pipelined network $\mathcal{S}'$ in $O(\frac{N}{p}D(\mathcal{S}))$ time. Further, by employing a layer-by-layer scan of the nodes in $\mathcal{S}'$, pairs of latches in each layer are combined into dummy comparators to obtain the augmented network $\mathcal{S}''$, and this process takes $O(\frac{N}{p}D(\mathcal{S}))$ time. It is easy to see that the task of constructing the line representation of $\mathcal{S}''$ from the graph representation of $\mathcal{S}''$ can also be carried out in $O(\frac{N}{p}D(\mathcal{S}))$ time. Finally, the greedy algorithm is performed on $\mathcal{S}''$ and this algorithm is essentially a Breath-First search, running in $O(\frac{N}{p}D(\mathcal{S}))$ time. Hence, the total time for generating a merge schedule from network $\mathcal{S}$ is bounded by $O(\frac{N}{p}D(\mathcal{S}))$.

It is interesting to note that, even if the MS schedule is available, the RMA needs $O(\frac{N}{p}D(\mathcal{S}))$ time to complete the task of sorting N elements. Thus, the time it takes the CU to compute an MS and the time needed by the network $\mathcal{T}$ to perform the sorting are perfectly balanced. In other words, the time complexity claimed in Theorem 3 also holds if the computation required for generating an MS is taken into account. It is very important to note that once available, the MS can be used to sort many problem instances.

The working space requirement by the control memory is proportional to $mD(\mathcal{S})$ words, each of $O(\log N)$ bits. Rather remarkably, the RMA does not require extra data memory space other than what is used for storing the input.

4 The Generalized Row-Merge Architecture

In a number of contexts, especially when the VLSI complexity of $\mathcal{T}$ is a concern, it is desirable to use an AT^2-optimal network as a parallel sorting device. The main goal of this section is to show that it is possible to design a sorting architecture that uses an AT^2-optimal sorting network as its parallel sorting device. As it turns out, the time performance of the new design, that we call the *Generalized Row-Merge Architecture* (GRMA, for short), is slightly better than that of the RMA discussed in Section 2.

The GRMA uses a sorting network $\mathcal{T}$ of fixed I/O size p with inputs $I_1, I_2, \ldots, I_p$ and outputs $O_1, O_2, \ldots, O_p$. It has p constant-port data memory modules $M_1, M_2, \ldots, M_p$, collectively referred to as the data memory. For every k, $1 \leq k \leq p$, memory module M_k is connected to input I_k and to output O_k. In one parallel read operation, one memory row is read and supplied as input to $\mathcal{T}$; in one parallel write operation, the output of $\mathcal{T}$ is written back into one memory row. Just like in the RMA, the memory accesses and the operation of the sorting network $\mathcal{T}$ are controlled by the control unit (CU). There are, however, three major differences between the GRMA and the RMA.

1. The GRMA has p memory modules rather than $\frac{p}{2}$ memory modules.
2. The sorting network can sort $O(D(\mathcal{T}))$ memory rows in row-major order in $O(D(\mathcal{T}))$ time.
3. For simplicity, we assume that the GRMA operates in a different pipelining mode than the RMA. Specifically, a group of r memory rows are fed into the network in r consecutive time steps and, after sorting, the r rows are written back to memory in r consecutive time steps. After that, another group of r memory rows is fed into the network, and so on. This process is repeated until the elements in all groups of r memory rows are sorted. The value r is proportional to the depth $D(\mathcal{T})$ of $\mathcal{T}$. (We note here that by changing the control mechanism, the GRMA can also operate in fully pipelined mode, i.e., the network $\mathcal{T}$ can be fed continuously.)

We select for $\mathcal{T}$ Leighton's optimal sorting network [16] which is known to be AT^2-optimal. This network, which is a hardware implementation of the well-known Columnsort

```
procedure Merge_Two_Super-rows($(a, b)$)
begin
    /* feeding phase */
    for $i = 1$ to $\frac{\log q}{2}$ do
        Read row $\frac{(a-1)\log q}{2} + i$ and feed this row to the input of $\mathcal{T}$
    endfor
    for $i = 1$ to $\frac{\log q}{2}$ do
        Read row $\frac{(b-1)\log q}{2} + i$ and feed this row to the input of $\mathcal{T}$
    endfor
    /* idling */
    for $i = 1$ to $D(\mathcal{T}) - \log q$ do no-op
    /* clearing phase */
    for $i = 1$ to $\frac{\log q}{2}$ do
        Write the output of $\mathcal{T}$ into row $\frac{(a-1)\log q}{2} + i$
    endfor
    for $i = 1$ to $\frac{\log q}{2}$ do
        Write the output of $\mathcal{T}$ into row $\frac{(b-1)\log q}{2} + i$
    endfor
end
```

Fig. 9. The procedure `Merge_Two_Super-rows`.

algorithm, has I/O size $\frac{q}{\log q}$ and depth $c \log q$, where c is a constant greater than 1. Two designs were proposed in [16]: one with a value of c significantly smaller than that of the other. Leighton's sorting network sorts an array of size $\log q \times \frac{q}{\log q}$ in row-major order in a pipelined fashion. Specifically, in each of the first $\log q$ steps, $\frac{q}{\log q}$ elements are fed into the network and, after $c \log q$ steps, these elements emerge, sorted, at the output of the network in $\log q$ consecutive steps.

Let q be such that $p = \frac{q}{\log q}$. We partition the $\frac{N}{p}$ memory rows into $m = \frac{2N}{p \log q}$ *super-rows*, each containing $\frac{\log q}{2}$ consecutive memory rows. That is, the ith super-row consists of memory rows $\frac{(i-1)\log q}{2} + 1$ through $\frac{i \log q}{2}$. The operation of the GRMA is partitioned into iterations, with two super-rows being sorted in each iteration. Each iteration consists of two phases:

- a *feeding phase*, during which two super-rows that contain $\log q$ memory rows are fed continuously into the sorting network, and
- a *clearing phase*, in which the elements in the sorting network are "drained" out.

Let (a, b) be an ordered pair of super-row numbers. The procedure `Merge_Two_Super-rows`(a, b), whose details are given in Fig. 9, performs the merge-split operation on the super-rows a and b.

We use a layered sorting network $\mathcal{S}$ of I/O size $m = \frac{2N}{p \log q}$ to obtain a merge schedule

$$MS = < (a_1, b_1), (a_2, b_2), \cdots, (a_s, b_s) >,$$

where $s = C(\mathcal{S})$, as follows: Let C_i be the list of comparators in layer L_i of $\mathcal{S}$. We concatenate these lists to obtain a sequence C of comparators such that C_i precedes C_{i+1}. Let $C = (c_{k_1}, c_{k_2}, \cdots, c_{k_s})$, $s = C(\mathcal{S})$, be the sequence of comparators obtained from $\mathcal{S}$. Based on C and on the line representation of $\mathcal{S}$, we obtain a merge schedule MS. Guided by MS, the sorting process proceeds in nonoverlapping iterations, each consisting of a call to procedure `Merge_Two_Super-rows`(a, b) to perform a merge-split operation on two super-rows specified by the (a, b) pair in MS.

By the compare-exchange/merge-split principle, the GRMA sorts N elements correctly. Since each iteration takes $O(\log q)$ time, the task of sorting N elements on the GRMA using this MS takes $C(\mathcal{S}) \log \Pi)$ time. Using the layered AKS network as $\mathcal{S}$, we obtain a valid MS of length $O(\frac{N}{p \log q} \log \frac{N}{p \log q})$ that can be used to sort N elements on the GRMA in $O(\frac{N}{p} \log \frac{N}{p \log p})$ time.

An argument similar to that used in the proof of Theorem 1 shows that every algorithm that sorts N elements on the GRMA requires at least $\Omega(\frac{N}{p} \log \frac{N}{p \log p})$ time. To summarize our findings, we state the following result:

	M_1	M_2	M_3	M_4
Row 1	15	12	9	4
Row 2	7	3	8	6
Row 3	22	14	21	37
Row 4	17	12	45	19
Row 5	26	1	32	11
Row 6	40	41	81	33
Row 7	51	6	16	21
Row 8	44	5	31	13

(a)

	M_1	M_2	M_3	M_4
Row 1	3	6	8	9
Row 2	12	15	21	45
Row 3	17	19	22	37
Row 4	4	7	12	14
Row 5	1	11	26	32
Row 6	33	40	41	81
Row 7	21	31	44	51
Row 8	5	6	13	16

(b)

	M_1	M_2	M_3	M_4
Row 1	3	6	8	9
Row 2	4	7	12	12
Row 3	17	19	22	37
Row 4	14	15	21	45
Row 5	31	32	44	51
Row 6	33	40	41	81
Row 7	1	11	21	26
Row 8	5	6	13	16

(c)

	M_1	M_2	M_3	M_4
Row 1	3	4	6	7
Row 2	8	9	12	12
Row 3	14	15	17	19
Row 4	21	22	37	45
Row 5	41	44	51	81
Row 6	31	32	33	40
Row 7	13	16	21	26
Row 8	1	5	6	11

(d)

	M_1	M_2	M_3	M_4
Row 1	3	4	6	7
Row 2	8	9	12	12
Row 3	13	14	15	16
Row 4	1	5	6	11
Row 5	41	44	51	81
Row 6	31	32	33	40
Row 7	17	19	21	26
Row 8	21	22	37	45

(e)

	M_1	M_2	M_3	M_4
Row 1	3	4	6	7
Row 2	1	5	6	8
Row 3	13	14	15	16
Row 4	9	11	12	12
Row 5	17	19	21	26
Row 6	21	22	31	32
Row 7	41	44	51	81
Row 8	33	37	40	45

(f)

	M_1	M_2	M_3	M_4
Row 1	1	3	4	5
Row 2	6	6	7	8
Row 3	9	11	12	12
Row 4	13	14	15	16
Row 5	17	19	21	21
Row 6	22	26	31	32
Row 7	33	37	40	41
Row 8	44	45	51	81

(g)

Fig. 10. An example illustrating the working of the RMA.

Theorem 4. *The Generalized Row-Merge Architecture that uses an AT^2-optimal network of I/O size p as the parallel sorting device sorts N elements, $N \geq O(p \log p)$, in $\Theta(\frac{N}{p} \log \frac{N}{p \log p})$ time.*

Notice that in the case of the GRMA, the computation of a merge schedule does not require an augmented network. The length of the merge schedule is somewhat shorter because of using a network $\mathcal{S}$ of smaller I/O size and depth, and without dummy comparators. Again, no extra data memory is required other than what is used for storing the input.

5 Conclusions and Open Problems

The main motivation for this work was provided by the observation that, up to now, sorting networks of fixed I/O size p had only been used to sort a set of p elements. Real-life applications, however, require sorting arbitrarily large data sets. Rather surprisingly, the important problem of using a fixed I/O size sorting network in such a context has not been addressed in the literature.

Our main contribution is to propose a simple sorting architecture whose main feature is the pipelined use of a sorting network of fixed I/O size p for sorting an arbitrarily large data set of N elements. A noteworthy feature of our design is that it does not require extra data memory space, other than what is used to store the input. As it turns out, the time performance of our design, that we call the Row-Merge Architecture (RMA) is virtually independent of the cost and depth of the underlying sorting network. Specifically, we showed that by using the RMA N elements can be sorted in $\Theta(\frac{N}{p}\log\frac{N}{p})$ time, without memory access conflicts. In addition, we showed how to use an AT^2-optimal sorting network of fixed I/O size p to construct a similar architecture, termed Generalized Row-Merge Architecture (GRMA) that sorts N elements in $\Theta(\frac{N}{p}\log\frac{N}{p\log p})$ time.

By removing the restriction on the rigid memory access schemes of RMA and GRMA, better performance can be achieved. Recently, we obtained a new realistic hardware-algorithm for sorting an arbitrary number of N elements using a sorting network of fixed I/O size of p in $\Omega\left(\frac{N\log N}{p\log p}\right)$ time. The architecture and control algorithm of this new design are much more complicated than the ones presented in this paper. Due to the allowed more flexible, yet regular, memory access patterns, this new architecture-algorithm combination achieves the best possible performance according to the lower bound result of [7] on sorting using a p-sorter. A preliminary report on this new result can be found in [29].

The best performance of the designs proposed in this paper is proportional to the depth of the AKS network, which is used to construct merge schedules. The constant associated with the depth complexity of the AKS network is too large to be considered practical. However, our results reveal the potential of row-merge based simple sorting architectures.

Along this line of thought, a long-standing open problem is to obtain a realistic sorting network of logarithmic depth. It is equally important to discover a network of depth $c\log m\log\log m$, where m is the network I/O size and c is a small constant. Such networks are useful for deriving practically short merge schedules.

Appendix

A Worked Example

The main goal of this Appendix is to illustrate, by using an example, how the RMA works. For this purpose, we take as $\mathcal{S}$ Batcher's bitonic sorting network of I/O size $p = 8$ featured in Fig. 1 whose line representation is depicted in Fig. 5b. The sorting device $\mathcal{T}$ is again Batcher's bitonic sorting network of I/O size $p = 8$ featured in Fig. 1.

The input is a collection of $N = 32$ items stored in $\frac{p}{2}$ memory modules, as illustrated in Fig. 10a. From the line representation of $\mathcal{S}$ (which is identical to its augmented network $\mathcal{S}''$), we obtain the merge schedule presented in (1). Now, Fig. 10b-g illustrates the contents of the data memory after executing the data movement suggested by the subsequence of the MS in each of the six stages L_1–L_6 in Fig. 5b. We note that in the end, the memory rows are sorted in row-major order.

Acknowledgments

The authors wish to thank three anonymous referees for their exceptionally thorough comments and suggestions for improving the presentation of the material. We are grateful to one of the referees for pointing out reference [15] to us. Last, but not least, we are grateful to Prof. Palis for his timely and professional handling of our submission. This work was supported by the U.S. Office of Naval Research grant N00014-97-1-0526, U.S. National Science Foundation grants CCR-9522093 and ECS-9626215, and by Louisiana grant LEQSF (1996-99)-RD-A-16

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c\log n$ Parallel Steps," *Combinatorica,* vol. 3, pp. 1–19, 1983.

[2] S.G. Akl, *Parallel Sorting Algorithms.* New York: Academic Press, 1985.

[3] S.G. Akl, *Parallel Computation, Models and Methods.* Prentice Hall, 1997.

[4] M. Atallah, G. Frederickson, and S. Kosaraju, "Sorting with Efficient Use of Special-Purpose Sorters," *Information Processing Letters,* vol. 27, pp. 13–15, 1988.

[5] K.E. Batcher, "Sorting Networks and Their Applications," *Proc. AFIPS Conf.,* pp. 307–314, 1968.

[6] K.E. Batcher, "On Bitonic Sorting Networks," *Proc. Int'l Conf. Parallel Processing,* pp. 376–378, 1990.

[7] R. Beigel and J. Gill, "Sorting p Objects with a k-Sorter," *IEEE Trans. Computers,* vol. 39, pp. 714–716, 1990.

[8] D. Bitton, D.J. DeWitt, D.K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surveys,* vol. 13, pp. 287–318, 1984.

[9] G. Bilardi and F.P. Preparata, "A Minimum Area VLSI Network for $O(\log n)$ Time Sorting," *IEEE Trans. Computers,* vol. 34, pp. 336–343, 1985.

[10] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. Computers,"* vol. 27, pp. 84–87, 1978.

[11] R. Cole and A.R. Seigel, "Optimal VLSI Circuits for Sorting," *J. ACM,* vol. 35, pp. 777–809, 1988.

[12] R.L. Drysdale and E.H. Young, "Improved Divide/Sort/Merge Sorting Networks," *SIAM J. Computing,* vol. 3, pp. 264–270, 1975.

[13] J.-W. Jang and V.K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *J. Parallel and Distributed Computing,* vol. 25, pp. 31–41, 1995.

[14] D.E. Knuth, *The Art of Computer Programming,* vol. 3. Reading, Mass.: Addison-Wesley, 1973.

[15] J.-D. Lee and K.E. Batcher, "Bitonic Sorting with a Fixed Number of Processors," a manuscript, 1998.

[16] F.T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Computers,* vol. 34, pp. 344–354, 1985.

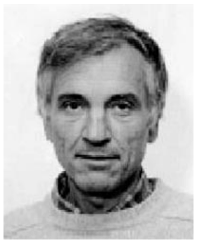
[17] R. Lin, S. Olariu, J. Schwing, and J. Zhang, "Sorting in O(1) Time on a Reconfigurable Mesh of Size $p \times n$," *Parallel Computing: From Theory to Sound Practice, Proc. EWPC '92,* pp. 16–27, 1992.

[18] D. Nassimi and S. Sahni, "Parallel Permutation and Sorting Algorithms and a New Generalized Sorting Network," *J. ACM,* vol. 29, pp. 642–667, 1982.

[19] M. Nigam and S. Sahni, "Sorting n Numbers on $n \times n$ Reconfigurable Meshes with Buses," *J. Parallel and Distributed Computing,* vol. 23, pp. 37–48, 1994.

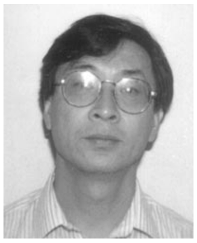
[20] S. Olariu and J. Schwing, "A New Deterministic Sampling Scheme with Applications to Broadcast Efficient Sorting on the Reconfigurable Mesh," *J. Parallel and Distributed Computing,* vol. 32, pp. 215–222, 1996.

[21] S. Olariu and S.-Q. Zheng, "Sorting N Items Using a p-Sorter in Optimal Time," *Proc. Eighth IEEE Symp. Parallel and Distributed Processing,* pp. 264–272, New Orleans, Oct. 1996.
[22] I. Parberry, "Current Progress on Efficient Sorting Networks," Technical Report CS-89-30, Dept. of Computer Science, Pennsylvania State Univ., 1989.
[23] B. Parker and I. Parberry, "Constructing Sorting Networks from k-Sorters," *Information Processing Letters,* vol. 33, pp. 157–162, 1989/1990.
[24] M.S. Paterson, "Improved Sorting Networks with $O(\log N)$ Depth," *Algorithmica,* vol. 5, pp. 75–92, 1990.
[25] D. Richards, "Parallel Sorting: A Bibliography," *ACM SIGACT News,* vol. 18, pp. 28–48, 1986.
[26] D. Richards, "A Bibliography of Parallel Sorting," a manuscript, 1996.
[27] C.D. Thompson, "The VLSI Complexity of Sorting," *IEEE Trans. Computers,* vol. 32, pp. 1,171–1,184, 1983.
[28] S.S. Tseng and R.C.T. Lee, "A Parallel Sorting Scheme Whose Basic Operation Sorts n Elements," *Int'l J. Computer Information Science,* vol. 14, pp. 455–467, 1985.
[29] S. Olariu, M.C. Pinotti, and S.Q. Zheng, "An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device," *Proc. 10th IASTED Int'l Conf. Parallel and Distributed Computing and Systems,* pp. 38-44, 1998.



Stephan Olariu received the MSc and PhD degrees in computer science from McGill University, Montreal, in 1983 and 1986, respectively. In 1986, he joined Old Dominion University, where he is a professor of computer science. Dr. Olariu has published extensively in various journals, book chapters, and conference proceedings. His research interests include image processing and machine vision, parallel architectures, design and analysis of parallel algorithms, computational graph theory, computational geometry, and mobile computing. He is an associate editor of the *International Journal of Computer Mathematics* and *IEEE Transactions on Parallel and Distributed Systems*, and serves on the editorial boards of the *Journal of Parallel and Distributed Computing*, *VLSI Design*, and the *International Journal of Foundations of Computer Science*. He is a member of the IEEE Computer Society.

M. Cristina Pinotti received the Dr. degree *cum laude* in computer science from the University of Pisa, Italy, in 1986. Since 1987, she has been a researcher with the National Council of Research at the Istituto di Elaborazione dell'Informazione, Pisa. Dr. Pinotti's research interests include computer arithmetic, residue number systems, VLSI layouts, design and analysis of parallel algorithms, parallel data structures, and multiprocessor interconnection networks. She is a member of the IEEE Computer Society.



Si Qing Zheng received the PhD degree in computer science from the University of California, Santa Barbara, in 1987. After having been on the faculty at Louisiana State University for eleven years, he joined the University of Texas at Dallas in 1998, where is he currently a professor of computer science. Dr. Zheng's research interests include algorithms, computer architectures, networks, parallel and distributed processing, telecommunication, and VLSI design. He was the program committee chairman of the Eighth International Conference on Computing and Information, the program committee co-chairman of the 10th International Conference on Parallel and Distributed Computing Systems, and the program committee vice chairman of the Second International Conference on Parallel and Distributed Computing Networks. He is an associate editor of several computing journals and is a senior member of the IEEE.