VLSI DESIGN
2000, Vol. 11, No. 2, pp. 137–147
Reprints available directly from the publisher
Photocopying permitted by license only

An Efficient Parallel VLSI Sorting Architecture

YANJUN ZHANG[a] and S. Q. ZHENG[b,*]

[a]*Research Group, Sabre, Inc., 1 East Kirkwood Blvd., MD 7390, Southlake, TX 76092;* [b]*Department of Computer Science, Box 830688, MS EC31, University of Texas at Dallas, Richardson, TX 75083-0688*

(Received 10 January 1999; In final form 6 December 1999)

We present a new parallel sorting algorithm that uses a fixed-size sorter iteratively to sort inputs of arbitrary size. A parallel sorting architecture based on this algorithm is proposed. This architecture consists of three components, linear arrays that support constant-time operations, a multilevel sorting network, and a termination detection tree, all operating concurrently in systolic processing fashion. The structure of this sorting architecture is simple and regular, highly suitable for VLSI realization. Theoretical analysis and experimental data indicate that the performance of this architecture is likely to be excellent in practice.

Keywords: Sorting network, VLSI, special-purpose architecture, systolic array, circuit layout

1. INTRODUCTION

The current VLSI technology has made it possible to directly implement parallel algorithms in hardware. Algorithm-structured chips are becoming the basic building blocks in the new generation of high-performance computing systems. This paper addresses the design and analysis of a new parallel sorting algorithm, and its VLSI implementation based on a sorting network.

There has been much research in sorting networks and the related area of parallel sorting algorithms. For an early in-depth treatment of sorting networks, see [11]. For recent surveys, refer to [14, 15]. These networks are constructed using constant fan-in comparator switches or simply nodes. There are several parameters that are used to assess the performance of a sorting network. The *cost* of a sorting network is the total number of constant fan-in comparators of the network. The *depth* of a sorting network is the maximum number of comparators on a path from an input to an output. For example, the bitonic sorting network and the odd-even merge sorting network [5] both have cost $O(n \log^2 n)$ and depth $O(\log^2 n)$, where n is the network I/O size (or simply the network size). The time performance of a sorting network is *number of parallel steps* performed,

*Corresponding author. Tel.: (972) 883-2329, Fax: (972) 883-2349, e-mail: sizheng@utdallas.edu

which is usually the depth of the network. The AKS sorting network [1] is the first network of $O(\log n)$ depth and $O(n\backslash\log n)$ cost, but is impractical due to the huge constants associated with the asymptotical complexity. The optimality of a sorting network implemented in VLSI is measured by combining the VLSI layout area and the time performance of the network; the most commonly used is the AT^2 complexity. Many AT^2 optimal sorting networks under various word length models have been proposed (*e.g.* [6, 8, 9, 13]).

In this paper we first present a new parallel sorting algorithm, called *MM-sort*, that uses a fixed-size sorting network (inner sorter), to iteratively sort an input of arbitrary size. Then, we propose a sorting architecture based on MM-sort. This architecture consists of three types of components: linear arrays, a sorting network, say, bitonic sorting network, and a termination detection tree. The linear arrays are used to implement local queues, supporting constant-time operations. The sorting network is the inner sorter used by MM-sort. The termination detection tree is used to stop the sorting process as soon as all queues are sorted. These components operate concurrently in pipelined fashion. The resulting architecture has a simple, regular and expandable structure, highly suitable for VLSI realization.

We present both theoretical and experimental analysis of MM-sort. Let $2n$ be the input size of the fixed-size inner sorting network of depth $D(n)$. The number of local queues is n. Let N be the input size with at most $(\lceil N/n \rceil)$ elements per queue. We show in Section 2 that MM-sort sorts an input of size N in $O(N/n \log n)$ iterations. This upper bound is shown to be tight in the worst case. Given the delay $D(n)$ of the inner sorter, an input of size N is sorted in $O((N/n)\log n\,(D(n)+f\,(N/n)))$ time, where $O(f\,(N/n))$ is the time required for selecting a minimum and a maximum in a queue of (N/n) elements in a single iteration. The factor $O(f\,(N/n))$ for selection will be $O(1)$ when systolic arrays are used to implement the local queues. The delay of $D(n)$ by the inner sorter can be eliminated if pipelining is used. In Section 4, we show that a pipelined MM-sort takes $O((N/nD(n))\log n)$ iterations. As a result, with pipelining and systolic arrays, MM-sort sorts an input of size N in $O((N/n)\log n)$ time. We show that this upper bound is tight in the worst case. Our experimental results also indicate that MM-sort performs well with random inputs.

2. MM-SORT

There are n processors connected by a sorting network S of input size $2n$. sorting network of size 8 and depth 6. Each processor P_i is associated with a local memory Q_i. The N input elements are evenly distributed among these processor/memory pairs. We associate *n queues*, each managed by a dedicated processor, with the sorting network S such that for $1 \le i \le n$, the $(2i-1)$th and the $(2i)$th input/output channels of S are connected to the ith queue. The elements of an input are distributed evenly, but arbitrarily, among the n queues such that each queue has at most $\lceil N/n \rceil$ elements, where N is the number of elements of the input. We say that the queues are *globally* sorted if $x \le y$ for any x in the ith queue and y in the jth queue such that $i < j, 1 \le i,j \le n$.

The sorting process consists of two phases, the *global-sorting phase* in which the queues are globally sorted, and the *local-sorting phase* in which, in parallel, each queue is sorted by its dedicated processor, and concatenated to obtain the sorted output. The global-sorting phase proceeds in iterations. We assume that the compare-exchange operation performed by a comparator on two elements takes constant time. The code of MM-sort is given below, in which $\mathcal{S}$ is a sorting network of depth $D(n)$ with $2n$ inputs/outputs, Q_i is the ith queue, *deletemin*(Q_i) and *deletemax*(Q_i) are the operations that delete the minimum and the maximum from Q_i, respectively, and *insert*(Q_i, x) is the operation that inserts x into Q_i. Note that once the queues are globally sorted, the queues will remain globally sorted if more iterations are applied. A technicality arises with

$N < 2n$ that the initial distribution of the input elements may leave some queues empty or having a single element, making impossible to select two elements from that queue. One remedy is to add a duplicate to a queue with a single element and two special symbols larger than any input elements to an empty queue. The added elements are removed at the end.

Algorithm **MM-sort**
begin
  /* Initialization */
  for $1 \le i \le n$ **do in parallel**
    $Q_i \leftarrow$ at most $\lceil N/n \rceil$ elements of an input of size N
  /* Global-sorting Phase */
  repeat until the queues are globally sorted
    for $i = 1, 2, \ldots, n$ **do in parallel**
      $R_{i,1} \leftarrow deletemax(Q_i)$;
      $R_{i,2} \leftarrow deletemin(Q_i)$;
      $R_{i,1} \rightarrow$ the $(2i)$th input channel of $\mathcal{S}$;
      $R_{i,2} \rightarrow$ the $(2i+1)$th input channel of $\mathcal{S}$;
        wait for D steps; /* sorting network delay */
      $R_{i,1} \leftarrow$ the outcome of $(2i)$th output channel of $\mathcal{S}$;
      $R_{i,2} \leftarrow$ the outcome of $(2i+1)$th output channel of $\mathcal{S}$;
      insert $(Q_i, R_{i,1})$;
      insert $(Q_i, R_{i,2})$;
  /* Local-Sorting Phase */
  for $i = 1, 2, \ldots, n$ **do in parallel**
    Q_i sorts its elements, and outputs its sorted sequence.
end

The selection of minimum and maximum in a local queue can be accomplished in various ways. The *min-max heap* data structure [4] can be used to keep the elements of a queue in a priority queue, permitting the operations of deletemin, deletemax and insert to be performed in $O(\log(N/n))$ time. The construction of the min-max heap and the associated operations are performed *in-place*, *i.e.*, no extra space other than those storing the input elements are used.

3. ANALYSIS OF MM-SORT

We show that MM-sort globally sorts n queues within $\lceil N/n \rceil (\lceil \log_2 n \rceil + 1)$ iterations on any input of size N. We also show that this upper bound is tight in the worst case by constructing an input instance for which $N \log_2 n/3n$ iterations are needed. We use the "zero-one principle" to reduce the sorting problem with arbitrary input to the problem of sorting $0-1$ inputs. We show that if MM-sort globally sorts all $0-1$ input instances of N elements within t iterations, then it globally sorts all inputs of size N within t iterations. We would like to point out that our use of the zero-one principle differs from the its classical usage of proving correctness of a sorting network [11]. We apply this principle to prove the number of iterations required by an algorithm to obtain a correct global sorting result. Our zero-one principle is much stronger than the classical one.

For any $a > 0$, let $h_a(x)$ be 1 if $x \ge a$; otherwise, 0. Given a sequence $I = (x_1, x_2, \ldots, x_N)$, let $h_a(I)$ be the $0-1$ sequence $(h_a(x_1), h_a(x_2), \ldots, h_a(x_N))$. Let Q_i and $\hat{Q}_i$ be the ith queue of MM-sort on input I and $0-1$ input $h_a(I)$, respectively. In the initial distribution of I and $h_a(I)$, $h_a(x)$ is in $\hat{Q}_i$ if and only if x is in Q_i for any $x \in I$. Given $Q_i = \{x_1, x_2, \ldots, x_{m_i}\}$, we define $h_a(Q_i) = \{h_a(x_1), h_a(x_2), \ldots, h_a(x_{m_i})\}$.

LEMMA 1 *Suppose that MM-sort is executed on inputs I and $h_a(I)$ in parallel simultaneously. Then, after every iteration, $\hat{Q}_i = h_a(Q_i)$ for any $a > 0$.*

Proof We use induction on the number of iterations. Initially, the lemma holds by the distributions of I and $h_a(I)$. Assume that the lemma holds after k iterations. Suppose that $Q_i = \{y_{i,1}, y_{i,2}, \ldots, y_{i,m_i}\}$. Then, by induction hypothesis, $\hat{Q}_i = h_a(Q_i) = \{h_a(y_{i,1}), h_a(y_{i,2}), \ldots, h_a(y_{i,m_i})\}$. We may assume that $y_{i,1} = \max_{1 \le j \le m_i}\{y_{i,j}\}$ and $y_{i,2} = \min_{1 \le j \le m_i}\{y_{i,j}\}$ for $1 \le i \le n$. Then $Y = \{y_{1,1}, y_{1,2}, y_{2,1}, y_{2,2}, \ldots, y_{n,1}, y_{n,2}\}$ is the set of elements selected from all Q_i's at the $(k+1)$-th iteration. Let $Z = (z_1, z_2, \ldots, z_{2n})$ be the sorted sequence of Y. For any $a > 0$, $h_a(Y)$ is the set of elements selected

from all $\hat{Q}_i$'s at the $(k+1)$-th iteration, and $h_a(Z)$ is the sorted sequence of $h_a(Y)$. Hence, after one more iteration, $Q_i = \{z_{2i-1}, z_{2i}, y_{i,3}, \ldots, y_{i,m_i}\}$ and $\hat{Q}_i = \{h_a(z_{2i-1}), h_a(z_{2i}), h_a(y_{i,3}), \ldots, h_a(y_{i,m_i})\}$. The lemma holds after $k+1$ iterations. The induction is complete. ■

LEMMA 2 [Zero-one Prinicple] *If MM-sort globally sorts all* 0-1 *sequences of size N within t iterations, then it globally sorts any input sequence of size N within t iterations.*

Proof Suppose to the contrary that MM-sort does not sort an input I in t iterations. Then, after t iterations, there must exist two queues Q_i and Q_j and $x \in Q_i$ and $y \in Q_j$ such that $i < j$ but $x > y$. Let $a = (x+y)/2$. Then $h_a(x) = 1$ and $h_a(y) = 0$. On the 0-1 input $h_a(I)$, by Lemma 1, after t iterations, $h_a(x) = 1 \in \hat{Q}_i$ and $h_a(y) = 0 \in \hat{Q}_j$ with $i < j$. This contradicts the assumption that MM-sort sorts any 0-1 input sequence in t iterations. ■

THEOREM 1 *Using a sorting network of* $2n$ *inputs, MM-sort sorts its n queues globally in* $\lceil N/n\lceil \log_2 n\rceil + 1\rceil$ *iterations on any input sequence of size N.*

Proof By Lemma 2, we only need to show that MM-sort sorts the queues globally on any 0-1 input sequence of size N in $\lceil N/n\rceil$ $(\lceil \log_2 n\rceil + 1)$ iterations. Let I be an arbitrary 0-1 input sequence of size N. Let Z be the number of 0's in I. Assume that $0 < Z < N$; otherwise, the input is already sorted. Let $m = \lceil N/n\rceil$ and $k = \lceil Z/m\rceil$ for some $1 \le k < n$. Consider following two cases.

Case 1 $k = 1$. In this case, there are at most m 0's, and all 0's should go to Q_1. If Q_1 does not have all the 0's, then Q_1 will gain one more 0's per iteration, and at most m iterations is needed for all 0's to be transmitted into Q_1.

Case 2 $k > 1$. In this case, there are more than $m(k-1)$ 0's. Since each queue has at most m elements, at least k queues contain 0's. Thus, at least k 0's will be selected as minimums during the current iteration. Consider the changes in the first $\lfloor k/2\rfloor$ queues after this iteration. For $1 \le i \le \lfloor k/2\rfloor$, if Q_i contains some 1's before this iteration, then the number of 0's in queue Q_i will increase by at least one after this iteration; otherwise, no changes occur in Q_i. Hence, after m iterations, the first $\lfloor k/2\rfloor$ queues will contain all 0's. During each of the later iterations, each of the first $\lfloor k/2\rfloor$ queues contributes two 0's and also receives two 0's, thus incurs no changes. So we may consider the first $\lfloor k/2\rfloor$ queues as being "dropped out" of the process. The number Z' of 0's in the remaining queues is at most $mk - m\lfloor k/2\rfloor \le m\lceil k/2\rceil$. Let $k' = \lceil Z'/m\rceil \le \lceil k/2\rceil$. The factor k is dropped by half after every m iterations. Hence, there can be at most $m\lceil \log_2 k\rceil$ iterations in Case 2 before reaching to Case 1. After at most m iterations in Case 1, all 0's will be redistributed to $Q_1, Q_2, \ldots, Q_\alpha$ for some α, $1 \le \alpha \le n$, depending on the initial distribution of the input, such that the queues Q_β, if any, where $1 \le \beta \le \alpha - 1$, are full of 0's. This distribution of 0's means that the queues are globally sorted. Hence, after at most $m(\lceil \log_2 k\rceil + 1) \le \lceil N/n\rceil$ $(\lceil \log_2 n\rceil + 1)$ iterations, the queues are globally sorted. We remark that the given argument is applicable to the case the input elements are distributed unevenly among the queues by taking m to be the maximum of the queue sizes. ■

The following theorem shows that this upper bound is tight in the worst case.

THEOREM 2 *There exists an input instance of size N such that MM-sort requires* $\Omega((N/n)\log n)$ *iterations in the global-sorting phase.*

Proof Let $N = mn$, where m is a multiple of 3 and n is a power of 2. Consider the 0-1 input instance that each Q_i, $1 \le i \le n$, contains exactly $m/3$ 1's and $2m/3$ 0's. After first $m/3$ iterations, the first $n/2$ queues contain all 0's, thus dropping out of the process from now on, and each of the second $n/2$ queues is in the *reverse* state of having exactly $m/3$ 0's and $2m/3$ 1's. After another $m/3$ iterations, the second half of the remaining queues contain all 1's, thus dropping out of the process, and the first half return to their previous state of having $m/3$ 1's and $2m/3$ 0's. Thus each round of $m/3$ iterations

eliminates exactly a half of the remaining queues. Hence, $m \log_2 n/3$ iterations are required to sort the given input. ■

THEOREM 3 *Using a $D(n)$-depth sorting network that sorts $2n$ elements, MM-sort sorts any input of size N in time $O((N/n) \log n(D(n)+\log (N/n)))$ in the in-place fashion.*

Proof Let $m = N/n$. Each queue has $O(m)$ number of elements stored in a min-max heap. The initialization of the max-min heaps in parallel takes $O(m)$ time. With a $D(n)$-depth sorting network, it takes $O(D(n)+\log m)$ steps to perform a single iteration of the global-sorting phase. The additive factor $O(\log m)$ is for the delete and insert operations on the min-max heap. By Theorem 1, the number of iterations of the global-sorting phase is $O(m \log n)$. The time for the local-sorting phase is $O(m \log m)$, which is dominated by the stated bound. ■

4. PIPELINING

A distinct feature of MM-sort is that the iterations of using the inner sorter can be pipelined. The *pipelined* MM-sort is presented below. To ensure the sorting network can be used in a pipelined fashion, we consider a special class of sorting networks. A sorting network in this class is constructed using constant fan-in comparators, each is used at most once during the sorting process of its input elements. For examples, both the bitonic sorting network and the odd-even merging sorting network satisfy this condition. We further assume that a sorting network in this class is *leveled*, which means that the number of comparators in any input-output path is the same. The bitonic sorting network is leveled. Some non-leveled sorting networks, such as the odd-even merge sorting network, can be converted to leveled networks by properly inserting buffers as dummy comparators. Given a sorting network in this class, one can feed inputs into the network at each step without causing conflicts with the data fed into the networks at previous steps.

Pipelined MM-sort
begin
  repeat
    /* feeding phase */
    for $i = 1, 2, \ldots, n$ **do in parallel**
      Q_i performs $D(n)$ feeding steps;
    /* clearing phase */
    for $i = 1, 2, \ldots, n$ **do in parallel**
      Q_i performs $D(n)$ receiving steps;
    /* Testing */
    if the queues are globally sorted **then** STOP.
end

Let $D(n)$ be the depth of the inner sorting network. Each iteration of the pipelined MM-sort consists of a *feeding phase*, in which a group of $D(n)$ inputs, each containing $2n$ elements, are fed continuously into the sorting network, and a *clearing phase*, in which the elements in the sorting network are "drained" out. For each queue, a *feeding step* consists of the operations *deletemin*, *deletemax*, the operations of feeding its minimum and maximum into the sorting network, and a *receiving step* consists of the operations of receiving outputs from the sorting network and inserting the received elements into the queue. We shall assume that each feeding and receiving step can be performed in a constant time. As will be seen shortly in Section 5, with the linear systolic array implementation of the priority queues, a sequence of feeding and receiving steps can be performed in constant time per step.

For easy reference, the MM-sort without pipelining is called *non-pipelined* MM-sort. Tables I and II show the executions of the non-pipelined and the pipelined MM-sort on a 0-1 input of size $mn = 36$. We have $n = m = 6$ and $D(n) = 2$. The non-pipelined MM-sort takes 5 iterations. The pipelined MM-sort takes 2 iterations. The speedup by pipelining is 2.5, greater than $D(n) = 2$. This example tells us that the relation between the non-pipelined and pipelined MM-sort is non-trivial.

TABLE I An example of non-pipelined MM-sort

initial	1st iter.	2nd iter.	3rd iter.	4th iter.	5th iter.
000010	000010	000010	000010	000011	000011
001010	000010	000010	000011	000011	000111
001010	001010	000011	000111	000111	000111
101010	001011	001111	000111	000111	000111
101011	101111	001111	001111	000111	000111
111011	101111	101111	001111	001111	000111

TABLE II Pipelined MM-sort for the input of Table I

initial	1st iter.	2nd iter.
000010	000010	000011
001010	000010	000111
001010	000111	000111
101010	000111	000111
101011	001111	000111
111011	101111	000111

The main result of this section is that the number of iterations of the pipelined MM-sort is $O((N/n)\log n/D(n))$. As each iteration of pipelined MM-sort takes $O(D(n))$ steps, the total number of steps of the pipelined MM-sort is $O((N/n)\log n)$.

LEMMA 3 *If pipelined MM-sort globally sorts all* 0-1 *sequences of size N within t iterations, then it globally sorts any input sequence of size N within t iterations.*

Proof The proof is similar to the proof of Lemma 2. ■

LEMMA 4 *Let* N_0 *and* N_1 *be the number of* 0*'s and* 1*'s in the n queues, each having m elements.*

(i) *If* $N_0 \geq mn/2$, *then the number of queues containing at least* $\lceil m/3\rceil$ 0*'s is at least* $\lfloor n/4\rfloor$.
(ii) *If* $N_1 \geq mn/2$, *then the number of queues containing at least* $\lceil m/3\rceil$ 1*'s is at least* $\lfloor n/4\rfloor$.

Proof We only prove (i). A proof to (ii) is similar. Each queue has m elements. Suppose to the contrary that (i) does not hold. Let $n/4=\lfloor n/4\rfloor+y$ where $0\leq y<1$. Then at least $n-\lfloor n/4\rfloor+1=3n/4-y+1$ queues each has at most $\lceil m/3\rceil-1$ 0's. The remaining $\lfloor n/4\rfloor=n/4-y$ queues each has at most m 0's. Thus $N_0\leq(3n/4-y+1)(\lceil m/3\rceil-1)+(n/4+y-1)m\leq(3n/4-y+1)m/3+(n/4+y-1)m=nm/2-2m(1-y)/3<nm/2$, which contradicts the assumption that $N_0\geq nm/2$. ■

THEOREM 4 *Let n be the number of queues and* $D(n)$ *be the depth of the inner sorting network of size* $2n$. *Suppose that* $(N/n)\geq 2D(n)$. *The pipelined MM-sort sorts its n queues globally in* $O((N/nD(n))\log n)$ *iterations.*

Proof The proof is similar to that of Theorem 1. Let $N=mn$, and consider a particular iteration consisting of a feeding phase and a clearing phase. The feeding phase and the clearing phase are each of $D(n)$ steps. For $1\leq i\leq D(n)$, let m_i and M_i denote the n-vectors that are the minimums and the maximums, respectively, selected from the n queues at the ith feeding step of the feeding phase.

Consider the case that the number of 0's is no smaller than the number of 1's. By Lemma 4(i), at least $\lfloor n/4\rfloor$ queues contain more than $\lceil m/3\rceil$ 0's. Each queue loses two elements at each step of the feeding phase. Each n-vector m_i, $1\leq i\leq D(n)$, has at least $\lfloor n/4\rfloor$ 0's. At the ith step of the clearing phase, the $\lfloor n/4\rfloor$ or more 0's in m_i will be fed back to the first $\lfloor\lfloor n/4\rfloor 2\rfloor=\lfloor n/8\rfloor$ queues, two 0's for each. Thus, in the clearing phase, the first $\lfloor n/8\rfloor$ queues receive all 0's, and no 1's. On the other hand, at each step of the feeding phase, the first $\lfloor n/8\rfloor$ queues each lose at least one if they have 1's. Thus, after a single pipeline iteration, each of the first $\lfloor n/8\rfloor$ queues either loses all its 1's if it has at most $D(n)$ 1's, or reduces its number of 1's by $D(n)$. Hence, after $\lfloor m/D(n)\rfloor$ iterations, the first $\lfloor n/8\rfloor$ queues will have all 0's. In each subsequent iteration, each of the first $\lfloor n/8\rfloor$ queues will contribute $2D(n)$ 0's and receive $2D(n)$ 0's, thus remain unchanged. We may regard the first $\lfloor n/8\rfloor$ queues as being "dropped out" of the process. The similar situation occurs when the number of 1's is no smaller than the number of 0's. In this case, by Lemma 4(ii) and the same arguments, the last $\lfloor n/8\rfloor$ queues will have all 1's after $\lfloor m/D(n)\rfloor$

iterations. These queues remain unchanged in the subsequent iterations, and can be regarded as being "dropped out" of the process. Hence, after $O(m \log n/D(n))$ pipeline iterations, fewer than 8 consecutive queues may not be globally sorted. The problem is reduced to globally sort n queues with $n < 8$.

We now show that $O(m/D(n))$ pipeline iterations are sufficient to sort $n < 8$ queues. Assume that there are more 0's than 1's. By Lemma 4(ii), for $n \geq 3$, at least two queues have $\lceil m/4 \rceil$ or more 0's. Consider the first queue Q_1. Let $K = \min\{D(n), \lceil m/4 \rceil\}$. In each of the first K clearing steps, Q_1 will receive two 0's. On the other hand, in the corresponding first K feeding steps, Q_1 contributes at least K 1's or all its 1's. In the remaining clearing steps when $K < D(n)$, Q_1 will receive at least as many 0's as Q_1 contributed in the corresponding feeding steps. Thus the net gain in the number of 0's for Q_1 after each iteration is at least K. In $O(m/D(n))$ iterations, Q_i will contain all 0's and drop out of the process. In the case there are more 1's than 0's, the last queue will contain all 1's in $O(m/D(n))$ iterations, and drop out of the process. This process continues as long as $n \geq 3$ and the problem will be reduced to $n = 2$ after $O(m/D(n))$ iterations.

Consider the case $n = 2$ with two queues Q_1 and Q_2. We claim that if there more 0's than 1's in Q_1 and Q_2, then the 4 elements selected at each feeding step contain at least two 0's. Our claim guarantees that Q_1 will contain all 0's after $m/D(n)$ iterations, thus globally sorted. We observe two facts that imply our claim. First, if Q_1 or Q_2 contributes two 0's at some step, then the claim holds for the subsequent steps. Secondly, if Q_1 contributes a 1 (two 1's) at some step, then Q_2 must also contribute a 0 (two 0's) at the same step because there are more 0's than 1's. Hence, the total number of iterations required is $O(m \log n/D(n))$. ■

For the case that $m = N/n < 2D(n)$, the pipelining capacity of the sorting network cannot be fully utilized. One can feed $2\lfloor m/2 \rfloor$ groups of inputs, each contains $2n$ elements, into the network in a pipeline feeding phase. By an argument similar to the proof of Theorem 4, it is simple to show that the queues are globally sorted in $O(\log n)$ iterations. We shall exclude this degenerated case from further discussions.

COROLLARY 1 *The number of steps of the pipelined MM-sort for sorting an input of size N is* $O((N/n) \log n)$.

Proof Each pipeline iteration takes $D(n)$ steps. ■

The following theorem shows that the upper bound of Theorem 4 is tight in the worst case.

THEOREM 5 *There exists an input instance of size N such that the pipelined MM-sort requires* $\Omega((N/nD(n)) \log n)$ *iterations.*

Proof Let $N = mn$, where $m = 3pD(n)$ and n is a power of 2. Consider same input instance used in the proof of Theorem 2, *i.e.*, each Q_i, $1 \leq i \leq n$, contains exactly $m/3$ 1's and $2m/3$ 0's. After first p pipeline phases, the first $n/2$ queues contain all 0's, thus dropping out of the process from now on, and each of the second $n/2$ queues is in the *reverse* state of having exactly $m/3$ 0's and $2m/3$ 1's. After another p pipeline phases, the second half of the remaining queues contain all 1's, thus dropping out of the process, and the first half return to their previous state of having $m/3$ 1's and $2m/3$ 0's. Thus each round of p pipeline phases eliminates exactly a half of the remaining queues. Hence, $p \log_2 n$ pipeline phases are required to globally sort the given input. That is, for such an input the pipelined MM-sort requires $\Omega(p \log n) = \Omega((N/nD(n)) \log n)$ pipeline phases. ■

5. HARDWARE IMPLEMENTATION

Our hardware implementation of MM-sort consists of three components, linear systolic arrays for the local queues, an inner sorting network, and an AND-tree circuit for detecting termination condition.

5.1. Linear Systolic Arrays

A design of linear systolic array supporting efficient priority queue operations *insert*(Q, x), *delete*(Q, x) and *deletemin*(Q) is given by H. T. Kung [12] for which each of the operations *insert*(Q, x), *delete*(Q, x) and *deletemin*(Q) can be done in $O(1)$ time per operation for a sequence of such operations.

MM-sort requires both *deletemin* and *deletemax*. A pair of linear arrays are needed. One array organizes the elements of a queue as a priority queue supporting *deletemin* and the another array as a priority queue supporting *deletemax*. The *delete* operation, that deletes an arbitrary node, is also needed. The minimum element of min-array is removed by the *delete* operation from the max-array, and *vice versa*. Moreover, a linear array of size m can sort m elements in $O(m)$ time. When implemented in a linear array, each local queue can sort its m elements in $O(m)$ time in the local-sorting phase.

5.2. Inner Sorting Network

The primary criterion for selecting an inner sorting network is the VLSI feasibility. Systolic implementation of sorting networks have been considered in [2, 10]. The layout area of the bitonic merger, the shuffle sorting network [16], and the mesh connected odd-even transportation sorting network were analyzed and compared in [10]. The depth of the inner sorter is not significant. Our theoretical analysis and experimental data show that the use of pipelining eliminates the delay of inner sorter.

5.3. Detecting Circuit

The local queues are globally sorted if $M_1 \le m_2, M_2 \le m_3, \ldots, M_{n-1} \le m_n$, where M_i and m_i are the maximum and minimum of ith queue, respectively. An AND-tree with $n-1$ comparators at the leaf-level can serve as the circuit for detecting termination. The inputs to this detecting circuit are nothing but the selected elements M_i and m_i at each iteration. The systolic linear array, the sorting network and the detecting circuit operate concurrently, all in pipelined fashion. With this detecting circuit, the execution of MM-sort terminates as soon as the input is globally sorted. The early termination detection is a feature that is not possessed by many existing parallel sorting architectures.

5.4. Expandability

The logical structure of the VLSI layout for our sorting architecture is depicted in Figure 1. All the three major components, the linear systolic arrays, the latched sorting network, and the termination detection tree, have simple planar or near-planar structures, and therefore can be laid out on VLSI chip in a compact form. The total area for systolic queues is proportional to the input size in the word model. The area for the sorting network depends on the number of queues and the network structure. Logically, our design is size-independent. To sort more elements, we may increase the size of each linear systolic array, or the size of the sorting network, or both. The size of each array can be increased easily by adding an additional array with a constant number of wires for connection. Similarly, the sorting network also can be expanded easily without rearranging the existing interconnections. When the size of sorting network is enlarged, the termination detecting tree needs to be expanded accordingly.

We call the resulting VLSI architecture the *MM-sorter*. The linear arrays support constant insert and delete operations, and the latched sorting network support pipelining. The following claim follows directly from Theorem 1 and Corollary 1.

THEOREM 6 *When used in non-pipelined mode, MM-sorter sorts N elements in* $O((N/n)\, D(n) \log n)$ *time where* $D(n)$ *is the depth of the inner sorter. When used in pipelined mode, MM-sorter sorts N elements in* $O((N/n) \log n)$ *time.*

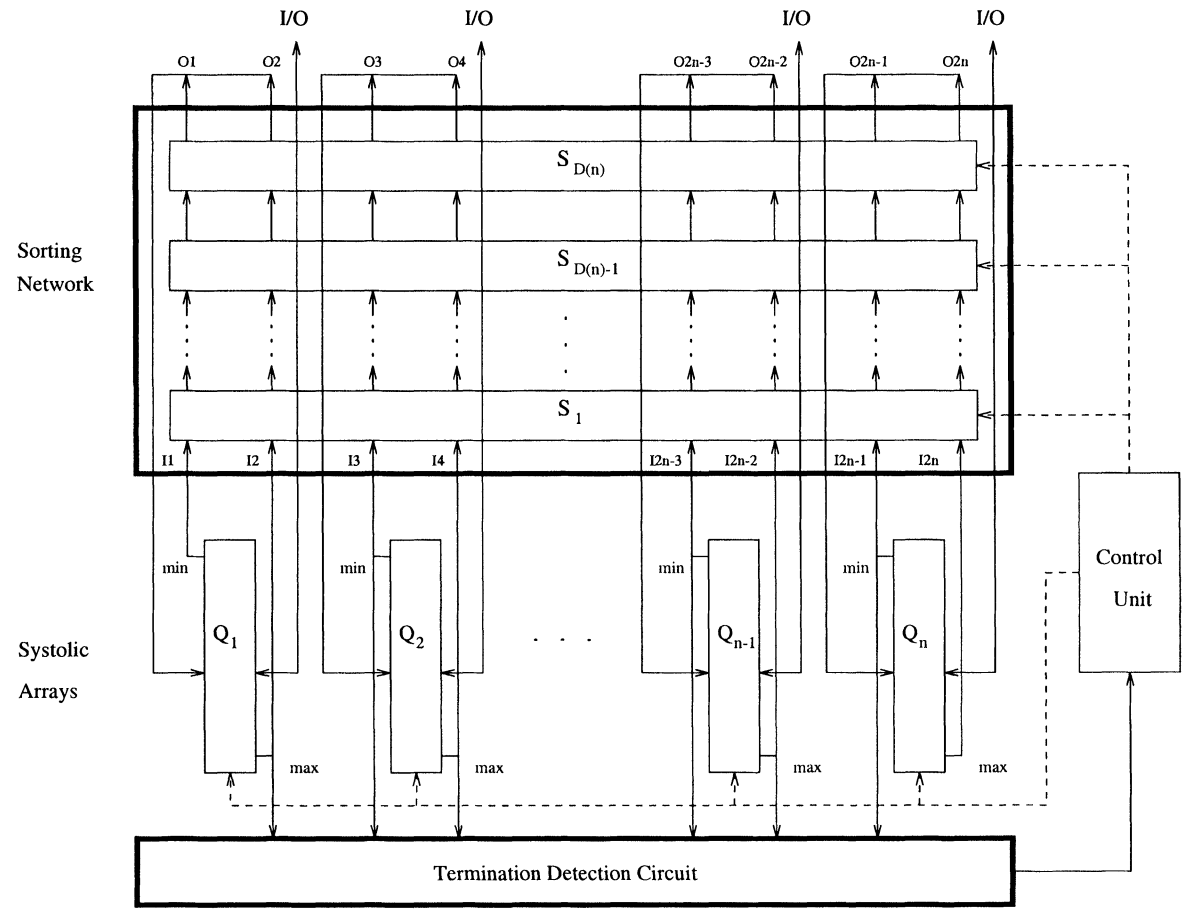

FIGURE 1 Logical structure of MM-sorter.

6. EXPERIMENTAL DATA

We present some experimental results of MM-sort on randomly generated inputs with a uniform distribution. Our data show that, on random inputs, the number of iterations of MM-sort is linear in the maximum size of a local queue, and the pipelining eliminates the delay of the sorting network.

Table III shows that the number of iterations of MM-sort required to globally sort the queues grows slowly as n increases. In particular, the average number of iterations in Table III never exceeds $3m$ for all instances, contrasting with the upper bound of $O(m \log n)$ given by Theorem 1. In Table III, m is the queue size and n is the number of queues. The input size is mn. We have chosen n to be a power of 2 up to $2^{11} = 2048$, and m a multiple of 20 up to 120. For each fixed choice of m and n, we run 10 samples and compute the average iterations of MM-sort over these 10 samples. The variances of these samples are remarkably small. For instance, for 10 samples of random inputs of size 60×2^{10} and queue size $m = 60$, the number of iterations are 143, 142, 146, 145, 144, 148, 144, 144, 142, 145, with a maximum of 148, a minimum of 142, and an average of 144.

Table IV presents experimental data for the pipelined MM-sort. The *speedup* of a single run is

TABLE III Average number of iterations of MM-sort over 10 runs of random inputs, where m is the size of the queues and n is the number of queues. The inner sorting network sorts $2n$ numbers

	$n=2$	4	8	16	32	64	128	256	512	1024	2048
$m=10$	6	8	11	12	15	17	19	21	23	25	27
20	9	15	21	25	29	34	37	41	44	48	52
40	19	31	43	51	59	65	73	80	88	96	101
60	30	47	65	77	87	99	109	122	132	143	153
80	40	63	85	103	120	135	148	165	179	192	207
100	52	77	110	131	149	168	187	208	225	242	262
120	58	93	131	156	184	205	224	248	272	292	314

TABLE IV Speedup of pipelined MM-sort over the non-pipelined MM-sort, averaged over 3 runs of random inputs, where $D(n)=(\log_2 n)^2$ is the depth of the sorting network that sorts $2n$ inputs. The inputs to the sorting network is pipelined

	$n=4$ $D=4$	8 9	16 16	32 25	64 36	128 49	256 64	516 81
$N/n=20$	**3.92	7.67						
40	*3.85	8.31	12.67					
60	*3.86	8.21	14.50	17.60				
80	3.92	*8.93	14.90	22.81	23.00			
120	3.92	8.75	15.13	23.33	33.44	32.38		
160	*3.95	*8.92	15.64	24.13	33.96	44.14	44.22	
200	*3.99	8.77	*15.58	23.92	35.07	43.67	53.63	59.17

the ratio of the non-pipelined version over the pipelined version. The numbers shown in Table IV are the speedup of the pipelined MM-sort over the non-pipelined MM-sort. The sorting network used here is of $2n$ inputs and depth $D(n)=(\log_2 n)^2$ (the depth of bitonic merging networks is $\Theta(\log^2 n)$). Our other choices of depths give similar results. To fill a pipeline of depth $D(n)$, each processor must have at least $2D(n)$ elements; when this condition is not met, the corresponding entries are left empty. The speedup given are averaged over 3 runs, as the variations between the runs are small. As we can see, the average speedup is remarkably close to the depth $D(n)$. The entries with a single asterisk (*) indicates that a perfect speedup $D(n)$ occurred with some runs, and the entry with double asterisks (**) indicates that a speedup greater than $D(n)$ has occurred.

7. CONCLUSION

We introduced a new parallel sorting algorithm, the MM-sort, for sorting a large set of elements using a fixed-size sorting network. We presented a systolic VLSI sorting architecture, the MM-sorter, based on MM-sort. The structure of MM-sorter is simple and regular, highly suitable for VLSI realization. We presented both theoretical analysis and the experimental data on the efficiency of MM-sort.

Theoretically, optimal sorting networks as those given in [1, 6, 13] should be used as the inner sorter to achieve the best theoretical performance in the non-pipelined mode. But the benefit in using such networks diminishes in the pipelined mode. We have shown, both theoretically and experimentally, that pipelining eliminates the delay of the inner sorter. As the depth of the inner sorter is not a factor in the pipelined mode, the more regularly structured sorting networks like bitonic sorters can be used as the inner sorter in the pipelined mode.

The experimental results in Section 6 show that MM-sort performs extremely well with random inputs. The number of iterations of MM-sort, with and without pipelining, is no more than $3(N/n)$ on all random inputs tested, eliminating the worst-case $\log n$ factor. An $O(N/n)$ bound on the number of iterations of MM-sort translates into an $O(N/n)$ running time of the MM-sorter, which is the best

possible. We left open the problem of analyzing the average running time of MM-sort.

Finally, we point out one interesting variation of MM-SORT, which we may call *ZZ-SORT* [17]. In ZZ-SORT, one element, instead of two, is selected from a queue at each iteration by alternating the two complimentary "zigzaging" rules: the odd-numbered queue selects its maximum and the even-numbered queue selects its minimum, and *vice versa*. We believe that ZZ-SORT has the same asymptotic behavior as MM-SORT.

References

[1] Ajai, M., Komlos, J. and Szemeredi, E. (1983). Sorting in $c \log n$ parallel steps, *Combinatorica*, **3**, 1–19.

[2] Afghahi, M. (1991). A 512 16-b bit-serial sorter chip, *IEEE Journal of Solid-State Circuits*, **26**, 1452–1457.

[3] Alnuweiri, H. M. (1993). A new class of optimal bounded-degree VLSI sorting networks, *IEEE Transactions on Computers*, **42**, 746–752.

[4] Atkinson, M. D., Sack, J. R., Santoro, N. and Strothotte, T. (1986). Min-max heaps and generalized priority-queues, *Communications of ACM*, **29**, 996–1000.

[5] Batcher, M. E. (1968). Sorting networks and their applications, *Proceedings of AFIPS Conference*, pp. 307–314.

[6] Bilardi, G. and Preparata, F. P. (1985). A minimum area VLSI network for $O(\log n)$ time sorting, *IEEE Transactions on Computers*, **C-34**, 336–343.

[7] Baudet, G. and Stevenson, D. (1978). Optimal sorting algorithms for parallel computers, *IEEE Transactions on Computers*, **C-27**, 84–87.

[8] Chien, M. V. and Oruc, A. Y. (1994). Adaptive binary sorting schemes and associated inter connection networks, *IEEE Transactions on Parallel and Distributed Systems*, **5**, 561–572.

[9] Cole, R. and Seigel, A. R. (1988). Optimal VLSI circuits for sorting, *Journal of ACM*, **35**, 777–809.

[10] Horiguchi, S. (1991). Hybrid systolic sorters, *Parallel Computing*, **17**, 997–1007.

[11] Knuth, D. E., *The art of computer programming*, **3**, Addison Wesley, 1973.

[12] Kung, H. T. (1980). The structure of parallel algorithms, *Advances in Computers*, **19**, 65–112.

[13] Leighton, T. (1985). Tight bounds on the complexity of parallel sorting, *IEEE Transactions on Computers*, **C-34**, 344–354.

[14] Parberry, I., Current progress on efficient sorting networks, *Tech. Rep. CS-89-30*, Dept. of Computer Science, Pennsylvania Univ., University Park, PA, 1989.

[15] Richards, D. (1986). Parallel sorting: A bibliography, *ACM SIGACT News*, **18**, 28–48.

[16] Stone, H. S. (1971). Parallel processing with the perfect shuffle, *IEEE Transactions on Computers*, **C-20**, 153–161.

[17] Zheng, S. Q., Calidas, B. and Zhang, Y. (1999). A general scheme for parallel in-place sorting, *Journal of Supercomputing*, **14**(1), 5–17.

Authors' Biographies

Yanjun Zhang received his Ph.D. in computer science from the University of California, Berkeley, in 1989. From 1990–1996, he was a faculty member of computer science at Southern Methodist University. Since 1997, he has been with Sabre, Inc. as a research scientist. His research interests include algorithm design and implementation, combinatorial optimization, parallel computation, and mathematical optimization in the airline industry.

S. Q. Zheng received his Ph.D. degree in computer science from the University of California, Santa Barbara, in 1987. After having been on the faculty of Louisiana State University for eleven years since 1987, he joined the University of Texas at Dallas, where he is currently a professor of computer science. Dr. Zheng's research interests include algorithms, computer architectures, networks, parallel and distributed processing, telecommunications, and VLSI design.