A hardware sorter for almost sorted sequences, with FPGA implementations

Naoaki Harada, Naoyuki Matsumoto, Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—Suppose that a sequence of sensing data with timestamps are transferred asynchronously. Some of sensing data may be delayed by some period of time and the sequence is not in proper increasing order of timestamps. A sequence of timestamps $t_0, t_1, \ldots, t_{n-1}$ is d-sorted if $t_i < t_j$ holds for all pairs of timestamps t_i and t_j such that $j-i \geq d$. Intuitively, a parameter d corresponds to the maximum period of delay and we can say that a d-sorted sequence is almost sorted if d is small. The main contribution of this paper is to show a hardware d-sorter that sorts a d-sorted sequence of timestamps, and to implement it in the FPGA. Our idea for a hardware d-sorter is to use a merge-based algorithm using FIFOs. This hardware algorithm takes $n + 2d + \log_2 d$ clock cycles using $\log_2 d$ comparators for a d-sorted sequence of n timestamps. Since $\Omega(n \log d)$ comparisons are necessary, it is cost optimal and attains optimal speedup. We also present the cyclic comparison method to reduce the number of bits in timestamps to be sorted. The experimental results show that the FPGA implementation is 5-10 time faster than a sequential d-sort program using a single CPU. Also, when $d = 2048$, our merge-based d-sorter is 2.59 times faster than a previously published hardware heap-based d-sorter, which takes $2n + 2d$ clock cycles.**

Keywords*—*parallel sorting algorithms; hardware algorithms; timestamp sorter; block RAMs

I. Introduction

An FPGA is a programmable logic device designed to be configured by the customer or designer by hardware description language after manufacturing. Since an FPGA chip maintains relative lower price and programmable features, it is widely used in those fields which need to update architecture or functions frequently such as image processing [1], [2]. Although the FPGA architecture is optimized for digital signal processing, it can be used for various general purpose computing [3], [4] and education [5]. Latest FPGA architectures consist of an array of Configurable Logic Blocks (CLBs), block RAMs, DSP slices, I/O pads, and interconnects [6], [7]. Since they work in parallel, FPGAs can be used to accelerate the computation.

The main purpose of this paper is to present an efficient FPGA implementation for sorting of an almost sorted sequence. Suppose that a sequence of sensing data with timestamps are input sequentially. Sensing data can be temperature data from thermometers, acceleration data from accelerometers, location data from GPS receivers, etc. and timestamps indicate time when they are gained. For example, sensing data are 32-bit single precision floating numbers with 64-bit unsigned integer representing a timestamp. If such sensing data are obtained and transferred asynchronously, some data may be delayed and sensing data in a sequence may not be in increasing order of timestamps. However, if the maximum amount of delay is guaranteed, we can say that a sequence of sensing data is almost sorted. We use a parameter d to represent the maximum delay as follows. A sequence $t_0, t_1, \ldots, t_{n-1}$ of n timestamps is *d-sorted* if $t_i < t_j$ holds for all pairs of timestamps t_i and t_j such that $j - i \geq d$. Intuitively, the value of d corresponds to the maximum of delay. The sorting of a d-sorted sequence of n timestamps can be done using a bottom-heavy heap implementing a priority queue. A heap is a binary tree based data structure, in which deletion of the minimum and insertion can be done by shifting down timestamps in the heap very efficiently [8]. Using a heap, soring of a d-sorted sequence of n timestamps can be done in $O(n \log d)$ time.

It is no doubt that sorting is one of the most important tasks in computer engineering, such as database operations, image processing, statistical methodology and so on. Hence, many sequential and parallel sorting algorithms have been studied in the past [9], [10]. The main purpose of this paper is to show an efficient hardware algorithm for sorting a d-sorted sequence. There are a lot of works that present efficient FPGA implementations for sorting of non-restricted sequences. For example, Marcelino *et al.* [11] implemented a FIFO-based merge sorter [12] and evaluated the performance on an FPGA. Koch and Torresen [13] presented FPGA implementations of of various parallel sorting algorithms. Mueller *et al.* [14] presented implementations of sorting networks [15]. However, sorting networks are very costly and require a lot of comparators. Marcelino *et al.* [16] show a simple architecture for parallel merge sort using one merge sorting unit. Since it repeats merging many times, the latency is quite large. Quite recently, Matsumoto *et al.* have presented a very efficient merge-based sorter [17], which minimizes the FIFO capacity used in FIFO-based merge sorter shown in [12]. Since the total capacity is only $n + O(\log n)$ and the capacity of n is necessary to complete sorting, their merge-based sorter is very close to optimal in terms of the total FIFO capacity. Since these FPGA implementations are designed for sorting of non-restricted sequences, it is not efficient to use them for sorting of almost sorted sequences. In [18], the heap-based d-sort is implemented in the FPGA. Each level of the binary heap is implemented as a dual-port block RAM in the FPGA. Since shifting down on the heap tree is performed on block RAMs corresponding to adjacent levels alternatively in parallel, each timestamp is output every two clock cycles. Thus, sorting of a d-sorted sequence with n timestamps takes at least $2n + 2d$ clock cycles.

In the merge-based d-sort, an input sequence of n timestamps is partitioned into $\frac{n}{d}$ subsequences of d timestamps each. Every subsequence is sorted independently, and adjacent subsequences are merged. We show a hardware algorithm to emulate the merge-based d-sort using FIFOs. The idea of our hardware merge-based d-sorter is to modify a parallel merge sorter [12] for sorting a d-sorted sequence. The resulting hardware d-sorter sorts a d-sorted sequence of n timestamps in $n + 2d + \log_2 d$ clock cycles. Since $d \ll n$ must hold from a practical point of view, the throughput our hardware merge-based d-sorter is almost twice as large as that of the hardware heap-based d-sorter shown in [18], which takes $2n + 2d$ clock cycles.

We also show an idea to reduce the number of bits stored in a timestamp. For example, if a C library function call clock() is used to get the value to be stored in timestamps, they are 64-bit unsigned integers which are incremented CLOCKS_PER_SEC times in a second. Usually, the value of CLOCKS_PER_SEC is 1000 or 1000000. If we use such 64-bit unsigned integers as they are, the timestamps must have 64 bits and 64-bit comparators are necessary to sort them. We show that if timestamps in an input sequence are enough frequent, the number of bits can be reduced. More specifically, we present *the cyclic comparison method* that allows us to reduce the number of bits in timestamps of such sequence. We also show that the cyclic comparison method can be implemented by a very simple combinational logic.

We have implemented our merge-based d-sorter with the cyclic comparison method in Virtex-7 Family FPGA XC7VX485T of VC707 Evaluation Kit [19]. We use distributed RAMs and block RAMs embedded in the FPGA to implement FIFOs. For efficient implementation, distributed RAMs are used for small FIFOs and block RAMs are used for large FIFOs. We assume that sensing data and timestamps have 18 bits each, because the word size of block RAM is a multiple of 18. Note that, since the cyclic comparison method is used, 18-bit timestamps may be sufficient to sort the sensing data correctly. The implementation results show that a 2048-sorter can be implemented using 1111 slices and 6 block RAMs on the FPGA. Since XC7VX485T has 75900 slices and 1030 block RAMs, the used hardware resources are only 1.5% slices and 0.58% block RAMs, respectively and the throughput is 182×10^6 sensing data per second.

We have also implemented the heap-based d-sort and the merge-based d-sort to run in a single Core i7-4790 (3.6GHz) CPU for reference. Since a word of CPU is 32-bit, we use sensing data and timestamps with 16 bits each. Their performances for 2048-sort are 17.6×10^6 and 20.1×10^6 sensing data per second, respectively. Thus, our FPGA implementation of the merge-based 2048-sort is 9 times faster than the CPU implementation, although the FPGA implementation uses very few hardware resources.

As we have mentioned, an FPGA implementation of the heap-based d-sorter has been presented in [18]. The performance was evaluated using a little older and smaller FPGA XC6VLX75T. We have also used the same FPGA for a fair comparison. The heap-based 2048-sorter and 65536-sorter in the FPGA runs in clock frequency 125 MHz and 100MHz, respectively. Since it outputs sensing data in every two clock cycles, the throughput is 62.5×10^6 and 50.0×10^6 sensing data per second, respectively. On the other hand, our implementation of the merge-based 2048-sorter and 65536-sorter in the FPGA runs in clock frequency 162 MHz and 95.1MHz, respectively. Since they can output sensing data in every clock cycle, the throughput is 162×10^6 and 95.1×10^6 sensing data per second, respectively. Thus, our implementation for the merge-based d-sorter achieves 1.90-2.59 times larger throughput that the heap-based d-sorter.

This paper is organized as follows. Section II shows two sequential algorithms for sorting a d-sorted sequences of n timestamps, which runs $O(n \log d)$ time and these algorithms are time optimal. In Section III, we show a hardware merge-based d-sorter and evaluate the performance. We introduce the cyclic comparison method to reduce the number of bits in timestamps in Section IV. In Section V we show our FPGA implementation of the merge-based d-sorter and experimental results. Section VI concludes our work.

II. Sequential sorting algorithms for a d-sorted sequence

The main purpose of this section is to show sequential algorithms for sorting a d-sorted sequence of n sensing data by timestamps. Since all algorithms presented in this paper are comparison-based, we simply consider sorting of timestamps and ignore sensing data.

We show two sequential algorithms, *the heap-based d-sort* and *the merge-based d-sort*, both of which run $O(n \log d)$ time. Although their ideas are quite natural, we show sophisticated versions that decrease the number of memory access and comparison operations. We also show that $\Omega(n \log d)$ time is necessary for comparison-based soring of a d-sorted sequence of n timestamps. Thus, these sequential algorithms are optimal.

We first define a d-sorted sequence formally. Let $T = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ be a sequence of n timestamps. To facilitate understanding, we assume that T is a permutation of $\langle 0, 1, \ldots, n-1 \rangle$. Thus, the sequence obtained after sorting T is $\langle 0, 1, \ldots, n-1 \rangle$. The results under this assumption can be applied to any non-restricted sequence, because all algorithms shown in this paper are comparison-based. A sequence T is *d-sorted* if $t_i < t_j$ holds for any i and j such that $j - i \geq d$. It should be clear that, if $t_i > t_j$ $(i < j)$ holds, then $j - i < d$ must be satisfied. Figure 1 shows an example of a 4-sorted sequence. Since $t_1 > t_4$, this sequence is not a 3-sorted sequence. Also, for any i and j such that $j - i \geq 4$, $t_i < t_j$ holds and thus, this sequence is a 4-sorted sequence.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	0	2	3	5	7	9	10	6	8	13	15	11	12	14

Fig. 1. An example of a 4-sorted sequence

We have the following basic lemma:

Lemma 1: For all i $(0 \leq i \leq n-1)$, one of $t_{i-d+1}, t_{i-d+2}, \ldots, t_{i+d-1}$ in a d-sorted sequence $T = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ must be i.

Proof: We assume that i equals t_j such that $j \leq i - d$ or $j \geq i + d$, and show that this assumption leads to a contradiction. Since T is d-sorted, all $j - d + 1$ timestamps

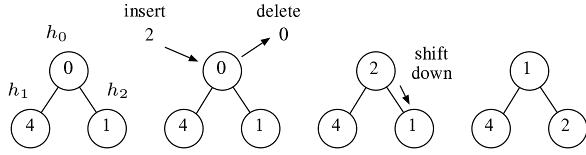

Fig. 2. The operations performed for replace-min-by(4) function call

$t_0, t_1, \ldots, t_{j-d}$ are smaller than $t_j = i$ and all $n-(j+d)$ timestamps $t_{j+d}, t_{j+d+1}, \ldots, t_{n-1}$ are larger than $t_j = i$. If $j \geq i+d$, it is not possible that at least $j-d+1$ ($> i$) timestamps smaller than i. Also, if $j \leq i-d$, it is not possible that at least $n-(j+d)-1$ ($> n-i$) timestamps are larger than i. Thus, $i-d < j < i+d$ must be satisfied. ■

We will show a simple sequential algorithm for sorting of d-sorted sequence with n timestamps. We use *a heap* [8] with capacity $d-1$ timestamps, which can be implemented by a full binary tree with $d-1$ nodes arranged in a 1-dimensional array of size $d-1$. Let $h_0, h_1, \ldots, h_{d-2}$ be elements of the 1-dimensional array of size $d-1$. The root h_0 stores the minimum of the d timestamps. Each h_i ($0 \leq i \leq \frac{d}{2}-1$) has two children h_{2i+1} and h_{2i+2}, and satisfies *the bottom-heavy heap property* such that the value stored in h_i is smaller than those stored in both children h_{2i+1} and h_{2i+2}. We use the following function calls handling a heap:
make-heap($t_0, t_1, \ldots, t_{d-2}$): Create a heap storing timestamps $t_0, t_1, \ldots, t_{d-2}$.
replace-min-by(t_i) Compare the t_i and the minimum timestamp in the heap. If t_i is smaller, t_i is returned. Otherwise, the minimum timestamp in the heap is replaced by t_i and is output.
delete-min(): Delete and returns the minimum timestamp in the heap.
After each of these calls, timestamps stored in heaps are shifted down to satisfy the bottom-heavy heap property, as illustrated in Figure 2.

Using these function calls for a heap, a d-sorted sequence with n timestamps can be sorted as follows:

[The heap-based d-sort]
make-heap($t_0, t_1, \ldots, t_{d-2}$);
for $i \leftarrow d-1$ to $n-1$ do
  output(replace-min-by(t_i));
for $i \leftarrow 0$ to $d-2$ do
  output(delete-min());

We will show that this algorithm outputs $0, 1, \ldots, n-1$ in turn correctly. After make-heap($t_0, t_1, \ldots, t_{d-2}$) is executed, $d-1$ timestamps are stored in a heap. From Lemma 1, 0 must be one of $t_0, t_1, \ldots, t_{d-1}$. Thus, either the minimum value stored in the heap or t_{d-1} is 0, and replace-min-by(t_{d-1}) returns 0. After that, replace-min-by(t_d) returns 1 because one of $t_0, t_1, \ldots, t_d$ is 1 from Lemma 1. Figure 2 illustrates the operations performed for replace-min-by(2) after make-heap(1,4,0) is executed. The minimum value 0 stored in the root h_0 is replaced by 2 and is output. After that, 2 "shifts down" in the heap to satisfy the bottom-heavy heap property. In general, it should be clear that replace-min-by(t_i) ($d-1 \leq i \leq n-1$) returns $i-(d-1)$. After replace-min-by(t_{n-1}) is executed, the heap stores timestamps $n-d+1, n-d+2, \ldots, n-1$. Thus, delete-min() outputs these timestamps one by one, and the heap-based d-sort performs sorting correctly.

Next, we will evaluate the running time of the heap-based d-sort. First, make-heap($t_0, t_1, \ldots, t_{d-2}$) is exactly the same as the first stage of the well-known heap sort. The heap of $d-1$ nodes can be created in $O(d)$ time by repeatedly merging heaps [8]. Each function call replace-min-by(t_i) and delete-min() and can be done in $O(\log d)$ time by shifting down operation of t_i. Hence, the total running time is $O(d) + (n-d+1) \cdot O(\log d) = O(n \log d)$, and we have,

Lemma 2: The heap-based d-sort sorts a d-sorted sequence with n timestamps in $O(n \log d)$ time.

We will show the merge-based sorting for a d-sorted sequence. Suppose that T is partitioned into $\frac{n}{d}$ subsequences $T_0, T_1, \ldots, T_{\frac{n}{d}-1}$ with d timestamps each. Let sort(T_i) be a procedure that sorts T_i by any sorting algorithm. We simply use merge sort [8] for sort(T_i). Also, let merge(T_i, T_{i+1}) be a procedure that merges two sorted sequences T_i and T_{i+1} and obtains one sorted sequence of $2d$ timestamps. The following algorithm sorts a d-sorted sequence:

[The merge-based d-sort]
for $i \leftarrow 0$ to $\frac{n}{d}-1$ do
  sort(T_i);
for $i \leftarrow 0$ to $\frac{n}{d}-2$ do
  merge(T_i, T_{i+1});

The merge-based d-sort sorts each subsequence independently. After that, adjacent two blocks are merged in turn. Figure 3 illustrates data movement by the merge-based d-sort for a 4-sorted sequence with 16 timestamps. After every subsequence is sorted, each pair of adjacent subsequences are merged from left to right.

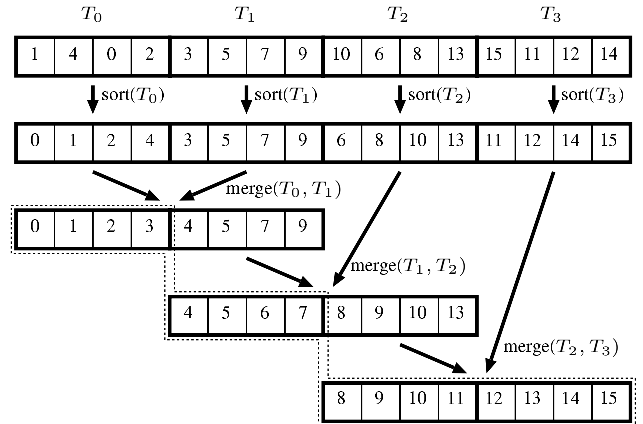

Fig. 3. The merge-based sorting for a 4-sorted sequence of 16 timestamps

Since T is d-sorted sequence and each subsequence has d timestamps, for all i and j such that $j-i \geq 2$, $T_i < T_j$ holds, that is, all timestamps in T_i are smaller than those of T_j. Thus, $T_0 \cup T_1$ includes all timestamps 0, 1, …, $d-1$, and after merge(T_0, T_1) is executed, T_0 stores 0, 1, …, $d-1$. In general, after merge(T_i, T_{i+1}) ($0 \leq i \leq \frac{n}{d}-2$) is executed, T_i stores $di, di+1, \ldots, di+d-1$.

Let us evaluate the running time of the merge-based d-sort. We use merge sort for each sort(T_i), which takes

$O(d \log d)$ time. Since we have $\frac{n}{d}$ blocks, the first for-loop takes $O(n \log d)$ time. After that, each merge(T_i, T_{i+1}) can be done in $O(d)$ time. More specifically, we compare the smallest timestamps in T_i and T_{i+1} and delete/output the smaller of the smallest timestamps. By repeating this operation $2d$ times, merge(T_i, T_{i+1}) can be done in $O(d)$ time. Thus, the second for-loop can be done in $O(n)$ time, and we have,

Lemma 3: The merge-based d-sort sorts a d-sorted sequence with n timestamps in $O(n \log d)$ time.

We will prove that both the heap-based d-sorting (Lemma 2) and the merge-based d-sorting (Lemma 3) are time optimal. It is well known that sorting of m data needs at least $\Omega(m \log m)$ comparisons [8]. We use this fact to prove $\Omega(n \log d)$-time lower bound. Suppose that a sequence T with n timestamps is partitioned into $\frac{n}{d}$ subsequences $T_0, T_1, \ldots, T_{\frac{n}{d}-1}$ with d timestamps each. We say that T is *d-block-sorted* if all elements in T_i are smaller than those of T_{i+1} for all i ($0 \leq i \leq \frac{n}{d} - 2$). Clearly, a d-block-sorted sequence is also a d-sorted sequence, because for any two timestamps t_i and t_j satisfying $j - i \geq d$, they are in different subsequences and $t_i < t_j$ holds. Hence, sorting of a d-block-sorted sequence is not harder than that of a d-sorted sequence and so the lower bound of sorting of a d-block-sorted sequence is also that of a d-sorted sequence. To sort a d-block-sorted sequence, each subsequence must be sorted. Since each subsequence has d timestamps, at least $\Omega(d \log d)$ time is necessary. Because a d-block-sorted sequence of n timestamps has $\frac{n}{d}$ subsequences, we have,

Lemma 4: At least $\Omega(n \log d)$ time is necessary to sort a d-sorted sequence with n timestamps.

From this lemma, the heap-based d-sort and the merge-based d-sort are time optimal.

III. A HARDWARE MERGE-BASED d-SORTER

The main purpose of this section is to show a FIFO-based parallel merge sorter for a d-sorted sequence.

Our architecture uses *a parallel merge sorter* [12] and *a sliding merger*. It basically emulates the merge-based d-sort shown for Lemma 3. A parallel merge sorter performs sort(T_i) and a sliding merger emulates merge(T_i, T_{i+1}).

We first show *a parallel merge sorter* briefly. *A d-merge-sorter*, which is a parallel merge sorter with parameter d, performs sort(T_i) for all i in a pipeline fashion, where every T_i has d timestamps. Figure 4 illustrates the architecture of an 8-merge-sorter. In general, a d-merge-sorter consists of $\log d$ mergers, 1-merger, 2-merger, 4-merger, ..., $\frac{d}{2}$-merger.

A k-merger ($k = 1, 2, 4, 8, \ldots$) has one input port and one output port and receives one timestamp from the input port and outputs one timestamp to the output port in every clock cycle. The input sequence of timestamps is partitioned into subsequences of k timestamps each and each subsequence is sorted. More specifically, an input sequence $T = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ are partitioned into $\frac{n}{k}$ subsequences $T_0, T_1, \ldots, T_{\frac{n}{k}-1}$ and each $T_i = \langle t_{i \cdot k}, t_{i \cdot k+1}, \ldots, t_{i \cdot k+k-1} \rangle$ ($0 \leq i \leq \frac{n}{k}-1$) is sorted. A k-merger has two FIFOs A and B that can store $k+1$ timestamps and k timestamps, respectively, as illustrated in Figure 4. Initially, both FIFOs are empty. First, all k timestamps in T_0 are enqueued in FIFO A one by one. After that, all k timestamps in T_1 are enqueued in FIFO B. Similarly, T_2 is enqueued in FIFO A and then T_3 is enqueued in FIFO B. This enqueue procedure is repeated until all timestamps in T are enqueued in FIFOs. At the same time, dequeue operation is performed. After the first timestamp t_k in T_1 is enqueued, we start dequeuing one of FIFOs A and B. Two timestamps in the heads of FIFOs A and B are compared and the smaller one is dequeued and sent to the output port. If FIFO B is empty, then FIFO A is dequeued. Also, if timestamps stored in the heads of two FIFOs are originated from different pairs, that from earlier pair is dequeued. During both enqueue and dequeue operations are performed, two FIFOs store totally $k + 1$ timestamps. It should be clear that FIFOs A and B may store $k + 1$ and k timestamps, respectively. For example, if all timestamps in T_0 are larger than those of T_1, then FIFO A will store all k timestamps in T_0 and the first timestamp in T_2. Thus, T_0 stores $k+1$ timestamps. Also, if all timestamps in T_0 are smaller than those of T_1, then FIFO B will store k timestamps.

Since k-merger starts outputting timestamps after $k + 1$ timestamps are input, the latency is $k+1$. Thus, the latency of d-merge-sorter is $(1+1)+(2+1)+(4+1)+\cdots+(\frac{d}{2}+1) = d + \log d - 1$, and we have,

Lemma 5: The d-merge-sorter performs sort(T_i) ($i \geq 0$) of the merge-based d-sort in latency $d + \log d - 1$.

From Figure 4, we can confirm that the latency of an 8-sorter is 10.

Next, we will show a d-sliding-merger, which performs merge(T_i, T_{i+1}) in the merge-based d-sort. A d-sliding-merger is very similar to a d-merger. Intuitively, if we use a k-merger for $k = d$, then merge(T_0, T_1), merge(T_2, T_3), merge(T_4, T_5), ... are performed in turn. On the other hand, a d-sliding-merger performs merge(T_0, T_1), merge(T_1, T_2), merge(T_2, T_3), ... in turn.

Figure 5 illustrates an architecture of an 8-sliding-merger. A d-sliding-merger has two FIFOs A and B with capacity d each. Similarly to a d-merger, all timestamps are enqueued in FIFOs A and B in turn. More specifically, timestamps in T_{2i} ($i \geq 0$) are enqueued in FIFO A and those in T_{2i+1} ($i \geq 0$) are eqnueued in FIFO B. After the first timestamp t_d in T_1 is enqueued, dequeue operation is started. In the dequeue operation, the heads of A and B are compared and smaller one is dequeued and output. Figure 5 also illustrates the timing chart of an 8-sliding-merger.

The reader may think that FIFO A may store $d + 1$ timestamps. However, it is sufficient for FIFO A to store d timestamps. Since the timestamps are d-sorted, the maximum timestamp in T_{2i+1} is larger than all timestamps in T_{2i}. Thus, while timestamps in T_{2i+2} is enqueued in FIFO A and those in T_{2i} in FIFO A is dequeued, FIFO B stores at least one timestamp in T_{2i+1} and cannot be empty. Since FIFOs A and B stores $d + 1$ timestamps totally, FIFO A never stores $d + 1$ timestamps. Hence, a d-sliding-merger emulates merge(T_i, T_{i+1}) for all i correctly, and we have,

Lemma 6: A d-sliding-merger performs merge(T_i, T_{i+1}) ($i \geq 0$) of the merge-based d-sort in latency $d + 1$.

Lemmas 5 and 6 combined, we have,

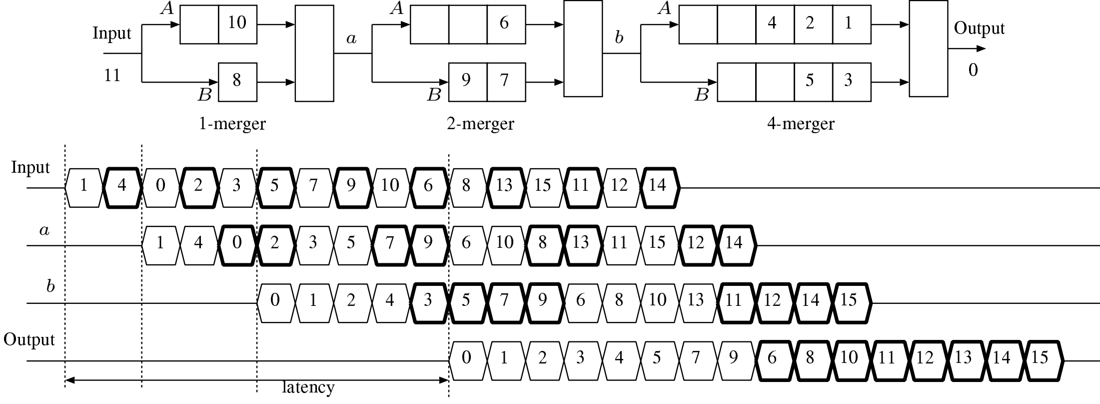

Fig. 4. The architecture of an 8-merge-sorter and the timing chart

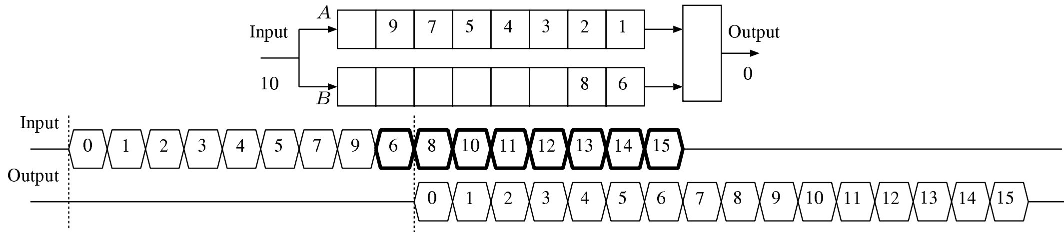

Fig. 5. The architecture of an 8-sliding-merger and the timing chart

Theorem 7: Sorting of a d-sorted sequence can be done in latency $2d + \log_2 d$.

Thus, if a d-sorted sequence has n time stamps, sorting can be done in $n + 2d + \log_2 d$ clock cycles. Since we use $\log_2 d + 1$ comparators, the total cost is $(n + 2d + \log d) \times (\log_2 d + 1) = O(n \log d)$. Hence, from Lemma 4, our merge-based d-sorter is optimal.

In [17], it has been shown that the total capacity of FIFOs used in k-mergers can be reduced by using more than two FIFOs with smaller capacity. We can use the same technique for d-sliding-merger. Two FIFOs of d-sliding-merger stores at most $d+1$ timestamps, while their total capacity is $2d$. We can reduce the total capacity by using more than two FIFOs with smaller capacity. Figure 6 illustrates an 8-sliding-merger using three FIFOs, which is simulating an 8-sliding-merger using two FIFOs illustrated in two FIFOs shown in Figure 5. The middle FIFO of the three may be used as FIFO A or FIFO B. In the figure, it is used as FIFO A. The 8-sliding-merger in Figure 5 uses two FIFOs with total capacity 16, while that in Figure 6 uses three FIFOs with total capacity 12. In general, if we use FIFOs with capacity M each, a d-sliding-merger can be implemented using $\frac{d}{M} + 1$ FIFOs with total capacity $(\frac{d}{M} + 1) \cdot M = d + M$.

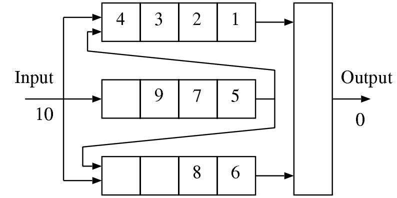

Fig. 6. 8-sliding-merger using three FIFOs

IV. The cyclic comparison method for a d-sorter

Suppose that sensing data with timestamps are given to the FPGA. If we use 64-bit unsigned integers returned by function call clock() as the values of timestamps, the FPGA needs to sort a sequence with 64-bit timestamps. The main purpose of this section is to show that we can remove the most significant bits and use the least significant bits for d-sort if some practical condition that we will show later is satisfied.

Suppose that the low-order 16 bits in 64 bits are used as timestamps. For example, for two 64-bit numbers

$$a = 1234\,5678\,9\mathrm{ABC}\,\mathrm{FFFF}$$
$$b = 1234\,5678\,9\mathrm{ABD}\,0001$$

in hexadecimal, $a < b$ holds. However, if the lower-order 16 bits $a' = \mathrm{FFFF}$ and $b' = 0001$ of them are compared, $a' > b'$ holds and the comparison results do not match. Thus, we introduce *the cyclic comparison*, which returns the correct comparison result if the difference of two numbers a and b are not large.

Suppose that for two distinct numbers a and b, let a' and b' be the low-order k bits of them, respectively. We assume that the difference of a and b is so small that $|a - b| < 2^{k-1}$ holds. Let $\delta = a - b$ and $\delta' = a' - b'$. Clearly, one of (A) $\delta = \delta'$, (B) $\delta = \delta' - 2^k$, or (C) $\delta = \delta' + 2^k$ holds. Since both $|\delta| < 2^{k-1}$ and $|\delta'| < 2^k$, we can determine if (A), (B), or (C) holds as follows:

CASE 1: If $\delta' < -2^{k-1}$ then (C) and $a > b$
CASE 2: If $-2^{k-1} < \delta' < 0$ then (A) and $a < b$ holds.
CASE 3: If $0 < \delta' < 2^{k-1}$ then (A) and $a > b$ holds.
CASE 4: If $2^{k-1} < \delta'$ then (B) and $a < b$ holds.
Note that, from $|\delta| \neq 2^{k-1}$, it is not possible that $|\delta'| = 2^{k-1}$. Suppose that $\delta' = a' - b'$ is computed using a k-bit subtractor and the resulting value is obtained as k-bit $s_{k-1}s_{k-2}\cdots s_0$. That is, this k-bit integer is the low-order k-bit of δ'. The reader should have no difficulty to confirm that, if CASE 1 or CASE 3, then $s_{k-1} = 0$. Let $k = 4$ and confirm this fact using an example. If $a' = 0101(= 5)$ and $b' = 1110(= 14)$ (CASE 1) then $\delta' = -9$ and $s = 0111$. If $a' = 0101(= 5)$ and $b' = 0010(= 2)$ (CASE 3) then $\delta' = 3$ and $s = 0011$. Also, if CASE 2 or CASE 4, then $s_{k-1} = 1$. Thus, we have,

Lemma 8: The equation $a > b$ holds if and only if $s_{k-1} = 0$.

From Lemma 8, we use value of s_{k-1} to decide if $a > b$. Hence, if $|a-b| < 2^{k-1}$ holds for all two timestamps compared in a sorting algorithm whose original values are a and b, the comparison of the low-order k-bits are sufficient sort data with such timestamps.

We first show that the heap-based d-sort never compares two timestamps with position difference more than $2d$. To confirm this fact, we assume that $T = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ to be sorted is a permutation of $\langle 0, 1, \ldots, n-1\rangle$ as before, and show that i and j such that $|i-j| \geq 2d$ are never compared. For a fixed i, let i' be an index such that $t_{i'} = i$. From Lemma 1, $i' > i - d$ holds. When replace-min-by($t_{i'}$) is performed, all values stored in the heap is larger than $i' - d$ from Lemma 1. From $i' - d > i - 2d$, replace-min-by($t_{i'}$) never performs comparison of i and $i-2d$. Hence, i and j such that $|i-j| \geq 2d$ are never compared.

Next, we will show that the merge-based d-sort never compare two timestamps with position difference $3d$ or more. For two timestamps i and j such that $j - i \geq 3d$, let i' and j' be indexes such that $t_{i'} = i$ and $t_{j'} = j$. From Lemma 1, $i' \leq i+d$ and $j' \geq j-d$. From $j-i \geq 3d$, we have $j'-i' \geq d$. Hence, $t_{j'}$ and $t_{i'}$ are in different subsequence in the merge-based d-sort, and they are never compared. Thus, we have,

Lemma 9: The heap-based d-sort and the merge-based d-sort never compare two timestamps with position difference more than $2d$ and $3d$, respectively.

Let $T' = \langle t'_0, t'_1, \ldots, t'_{n-1}\rangle$ be the sorted sequence of T. Suppose that T is sorted using the low-order k bits of timestamps. From this lemma, T can be sorted correctly by the merge-based d-sort if $t'_{i+3d} - t'_i < 2^{k-1}$ for all i.

V. The FPGA implementation and experimental results

The main purpose of this section is to show the running time and the performance of sorting for a d-sorted sequence.

For reference, we have implemented both sequential merge-based soring and heap-based sorting and evaluated the running time on an Intel Core i7-4790 (3.6GHz) processor. Although Intel Core i7 processor has multiple processor cores running in parallel, our implementations are sequential and do not use multiple cores. However, we can say that we can not expect 4 times or more acceleration ratio over our sequential implementation using a single core if we use 4 processor cores. For CPU implementation, we simply use 32-bit timestamps. Clearly, sorting for 32-bit timestamps can be applied as it is to sort of 16-bit sensing data with 16-bit timestamps.

We use a Virtex7-family FPGA XC7VX485T. It has 1030 *block RAMs*, which can be used as ring buffers for FIFOs. A single block RAM can be configured as one 36kbit block RAM or two 18kbit Block RAMs [7]. The standard data width of block RAMs is 36 bits. Hence, we assume that the input is a sequence of 18-bit sensing data with 18-bit timestamps. Thus, a 36kbit block RAM and a 18kbit block RAM can be used to implement FIFOs with 1024 and 512 timestamps, respectively. Also, larger FIFOs can be implemented using multiple block RAMs in an obvious way. Virtex-7 FPGAs also have a lot of Configurable Logic Blocks (CLBs), each of which has two slices [6]. For example, XC7VX485T has 37950 CLBs or 75900 slices. Each slice has four Look-Up Tables (LUTs), each of which is a $2^6 = 64$-bit memory. Multiple LUTs consistitute *a distributed RAM*, which is used to implement a FIFO of size less than 512. Also, in our implementation of our merge-based d-srter, 18-bit timestamps are compared by the cyclic comparison.

We have evaluated two types of implementations for each k-merger and each k-sliding-merger:
2-FIFO: Two FIFOs are used for all k-mergers and k-sliding-mergers. More specifically, two distributed RAMs are used for k up to 256, Two 18kbit RAMs (512×36-bit each) are used for $k = 512$ and $\frac{2k}{1024}$ 36kbit RAMs (1024×36-bit each) are used for $k \geq 1024$.
Optimal-FIFO: Similarly to the 2-FIFO implementation, we use block RAMs for $k \geq 512$. Further, we select the number of FIFOs used in each k-merger and each k-sliding-merger such that the hardware resource usage is minimized. For example, 8192-sliding-merger uses 9 FIFOs, each of which is implemented using a 36kbit block RAM. Note that in the 2-FIFO implementation, eight 36kbit block RAMs are used for each FIFO. Thus, sixteen 36kbit block RAMs are necessary to implement a 8192-sliding-merger by 2-FIFO.

Table I shows optimal selections of the number of FIFOs for each k. Merge-based d-sorter by Optimal-FIFO uses 1-merger, 2-merger, $\ldots$, $\frac{d}{2}$-merger, and d-sliding-merger with parameters shown in Table I.

Table II shows the performance for d-sort using Core i7 CPU and Virtex 7 FPGA. We have used d-sorted sequence of 64M $(= 2^{26})$ data for $d = 32$ to 65536. The performance of the merge-based d-sorting and the heap-based d-sorting on Intel Core i7 processor are evaluated by the throughput computed from the running time. We can see that the performance of two sequential sorting algorithms are almost the same. The merge-based d-sorting is slightly higher throughput for large d. This may be due to memory access locality. Access to timestamps in the FIFOs of the merge-based d-sorting is sequential. On the other hand, the access to timestamps in the heap of the heap-based d-sorting is stride and such stride access degrades memory cache performance.

Next, we compare the heap-based d-sorter [18] and our merge-based d-sorter. Table III shows the implementation results. Since the implementation results for the heap-based d-

TABLE I. THE OPTIMAL SELECTION OF THE NUMBER OF FIFOS

k		1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
k-merger	memory type	distributed RAMs									18kbit block RAMs	36kbit block RAMs						
	FIFOs	2	3	2	2	2	2	2	2	2	2	2	3	5	9	17	33	65
k-sliding -merger	memory type	distributed RAMs									18kbit block RAMs	36kbit block RAMs						
	FIFOs	-	-	-	-	-	2	2	2	2	2	2	3	5	9	17	33	65

TABLE II. THE PERFORMANCE FOR d-SORTING FOR CORE I7-4790 CPU AND VIRTEX 7 FPGA IMPLEMENTATIONS

d		32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
Heap-based d-sorting	10^6 data/sec	42.0	34.8	29.5	25.2	21.9	19.4	17.6	16.2	14.9	13.9	12.9	11.9
Merge-based d-sorting	10^6 data/sec	40.8	34.2	30.1	26.7	24.1	22.0	20.1	18.6	17.3	16.2	15.1	14.3
Merge-based d-sorter FPGA (2-FIFO)	10^6 data/sec	222	242	234	219	207	193	183	184	171	144	131	112
	slices (out of 75900)	358	491	604	784	889	957	1081	1219	1336	1557	1676	2035
	block RAMs (out of 1030)	0	0	0	0	1	3	7	15	31	63	127	255
	clock (MHz)	222	242	234	219	207	193	183	184	171	144	131	112
Merge-based d-sorter FPGA (Optimal-FIFO)	10^6 data/sec	232	241	247	224	207	188	182	166	171	164	147	132
	slices (out of 75900)	336	467	632	771	865	973	1111	1441	1963	2679	4136	6984
	block RAMs (out of 1030)	0	0	0	0	1	3	6	11	20	37	70	135
	clock (MHz)	232	241	247	224	207	188	182	166	171	164	147	132

sorter assumed Virtex6-Family FPGA XC6VLX75T, we have also used the same FPGA for a fair comparison. When $d = 2048$, our merge-based d-sorter uses less hardware resources and attains higher clock performance than the heap-based d-sorter, because the control logic of our merge-based sorter is simpler and distributed RAMs and block RAMs are used more efficiently. Note that the heap-based d-sorter needs two clocks to output each data, while our merge-based d-sorter can output a timestamp in every clock cycle. Thus, the throughput of our merge-based d-sorter is $\frac{162}{62.5} \approx 2.59$ times larger. On the other hand, when $d = 65536$, our merge-based d-sorter needs more hardware resources because our merge-based d-sorter needs many slices to control a lot of FIFOs. The throughput or merge-based d-sorter is $\frac{95.1}{50.0} \approx 1.90$ times larger. Also, the heap-based d-sorter shown in [18] does not support the cyclic comparison, while our merge-based d-sorter supports it.

TABLE III. THE PERFORMANCE OF d-SORTERS ON XC6VLX75T FPGA

d		2048	65536
Heap-based d-sorter FPGA [18]	10^6 data/sec	62.5	50.0
	slices (out of 11640)	1860	2484
	block RAMs (out of 156)	27	104
	clock (MHz)	125	100
Merge-based d-sorter FPGA (Optimal-FIFO)	10^6 data/sec	162	95.1
	slices (out of 11640)	1229	8161
	block RAMs (out of 156)	6	135
	clock (MHz)	162	95.1

VI. CONCLUSION

In this paper, we have presented the merge-based d-sorter, which can sort a d-sorted sequence of n timestamps in $n + 2d + \log_2 d$ clock cycles. It is about twice as fast as previously published hardware heap-based d-sorter, which takes $2n + 2d$ clock cycles. Also, the experimental results on a Virtex7 FPGA show that our merge-based d-sorter is 5-10 time faster than a sequential d-sort program using a single CPU and 1.9-2.6 times faster than the previously published heap-based d-sorter.

REFERENCES

[1] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, May 2003.

[2] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Trans. on Information & Systems*, vol. E88-D, no. 7, pp. 1350–1359, 2005.

[3] S. D. Kohale and R. W. Jasutkar, "Power optimization of GCD processor using low power Spartan 6 FPGA family," *International Journal of Conceptions on Electronics and Communication Engineering*, vol. 2, no. 1, pp. 1–6, June 2014.

[4] J. L. Bordim, Y. Ito, and K. Nakano, "Instance-specific solutions to accelerate the CKY parsing for large context-free grammars," *International Journal on Foundations of Computer Science*, vol. 15, no. 2, pp. 403–416, 2004.

[5] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an FPGA," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2008, pp. 723–728.

[6] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, Nov. 2014.

[7] ——, *7 Series FPGAs Memory Resources User Guide*, Nov. 2014.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[9] D. E. Knuth, *The Art of Computer Programming. Vol.3: Sorting and Searching*. Addison-Wesley, 1973.

[10] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press Inc., 1990.

[11] R. Marcelino, H. Neto, and J. M. Cardoso, "Sorting units for FPGA-based embedded systems," *Distributed Embedded Systems: Design, Middleware and Resources*, vol. 271, pp. 11–22, 2008.

[12] S. Todd, "Algorithm and hardware for a merge sort using multiple processors," *IBM Journal of Research and Development*, vol. 22, no. 5, pp. 509–517, Sept. 1978.

[13] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proc. of International Symposium on Field Programmable Gate Arrays*, 2011, pp. 45–54.

[14] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The International Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, Feb. 2012.

[15] K. Batcher, "Sorting networks and their applications," in *in Proceedings of the AFIPS Spring Joint Computer Conference 32*, 1968, pp. 307–314.

[16] R. Marcelino, H. C. Neto, and J. M. P. Cardoso, "Unbalanced FIFO sorting for FPGA-based systems," in *Proc. of International Conference on Electronics, Circuits, and Systems*, Dec. 2009, pp. 431 – 434.

[17] N. Matsumoto, K. Nakano, and Y. Ito, "Optimal parallel hardware k-sorter and top k-sorter, with FPGA implementations," in *Proc. of International Symposium on Parallel and Distributed Computing*, June 2015, pp. 138–147.

[18] W. M. Zabołotny, "Dual port memory based heapsort implementation for FPGA," in *Proc. SPIE, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, Oct. 2011.

[19] Xilinx Inc., *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*, 2014.