High-Performance Hardware Merge Sorter

Susumu Mashimo, Thiem Van Chu and Kenji Kise
Tokyo Institute of Technology
Email: mashimo@arch.cs.titech.ac.jp, thiem@arch.cs.titech.ac.jp, kise@c.titech.ac.jp

***Abstract*—State-of-the-art studies show that FPGA-based hardware merge sorters (HMSs) can achieve superior performance compared with optimized algorithms on CPUs and GPUs. The performance of any HMS is proportional to its operating frequency (F) and the number of records that can be output each cycle (E). However, all existing HMSs have a problem that F drops significantly with increasing E due to the increase of the number of levels of gates. In this paper, we propose novel architectures for HMSs where the number of levels of gates is constant when E is increased. We implement some HMSs adopting the proposed architectures on a Virtex-7 FPGA. The evaluation shows that an HMS of E = 32 operates at 311MHz and achieves 3.13x higher throughput than the state-of-the-art HMS.**

I. Introduction

Sorting is a fundamental operation used in a wide range of applications. Improving sorting performance is thus a critical issue that has been extensively studied for decades. Most of the efforts have been concentrated on enhancing sorting algorithms for general purpose CPUs [1], [2] or, more recently, GPUs [3], [4]. On the other hand, there has been interest in the hardware acceleration approach using FPGAs in recent years. State-of-the-art studies show that some hardware sorters can achieve superior performance compared with optimized sorting algorithms on CPUs and GPUs. For instance, the FPGA accelerator proposed by Casper and Olukotun [5] achieves a sorting throughput of 76.8 Gb/s, while recent studies of sorting on a computer with 16∼32 processor cores [1], [2] report sorting throughputs of 30 Gb/s at most, and 50 Gb/s at most with a GPU [3], [4].

Most of existing hardware sorters are based on the merge operation. We call such a hardware sorter a *hardware merge sorter*. The main component of these sorters is a *merge logic* that combines two sorted sequences into one sorted sequence. The basic merge logic outputs one record per cycle [6]. To achieve higher throughput, there are several studies of merge logics that output two or more records per cycle. We call the merge logic that can output E records per cycle an *E-record merge logic* ($E \geq 2$).

Each record to be sorted is divided into a key field and a data field, and the key field is used for comparing and reordering records. Throughout the paper, we simply say "the comparison of two records" for the comparison of the keys of the two records. Similarly, we refer to a record with a large/small key as "a large/small record". We assume that the sorting order is ascending.

The throughput of an E-record merge logic is the product of E and its operating frequency F. Thus, it is highly desirable to increase both E and F. However, existing merge logic architectures have a problem that the number of levels of gates increases with increasing E, and therefore F decreases. This problem makes it difficult to implement high-performance hardware merge sorters.

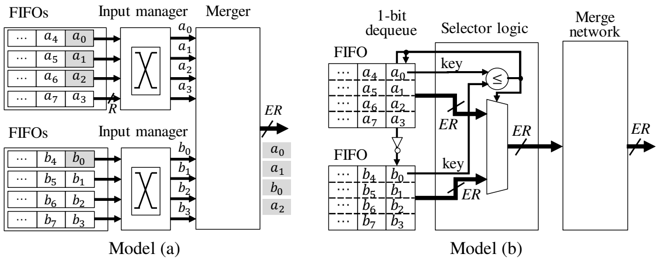

Fig. 1. Two models of E-record merge logics. In this figure, $E = 4$ and R is the record width. Our architecture is based on model (b).

We classify existing E-record merge logics into two models, model (a) and model (b), as described in Fig. 1. In this figure and all figures below, R is the bit width of a record. These merge logics merge two sequences:
$\mathbb{A} = \{a_0, a_1, \cdots, a_{n-1} \mid a_0 \leq a_1 \leq \cdots \leq a_{n-1}\}$ and
$\mathbb{B} = \{b_0, b_1, \cdots, b_{m-1} \mid b_0 \leq b_1 \leq \cdots \leq b_{m-1}\}$.

Firstly, we describe the behavior of the E-record merge logic in model (a). Previous work [7] adopts this model. It consists of two sets of FIFOs, two input managers and a merger.

Each set of FIFOs consists of E FIFOs whose data width is R. Two sorted sequences $\mathbb{A}$ and $\mathbb{B}$ are stored in these two sets of FIFOs and the records are located as shown in Fig. 1(a). The merger takes as its inputs two sets of sorted E records and outputs the E smallest records among them in sorted order.

For example, in Fig. 1(a), we suppose that
$a_0 \leq a_1 \leq b_0 \leq a_2 \leq b_1 \leq b_2 \leq a_3 \leq b_3$. Thus, E records $\{a_0, a_1, b_0, a_2\}$ marked with gray color are dequeued from their respective FIFOs. After that, E records at the front of each of two sets of FIFOs are $\{a_4, a_5, a_6, a_3\}$ and $\{b_4, b_1, b_2, b_3\}$. Here, E records at the front of each of two sets of FIFOs are not in sorted order and must be reordered because each of the two sets of E records input into the merger must be in sorted order.

The numbers of levels of gates in the merger and the two sets of FIFOs can be constant with respect to E. However, each input manager reorders the records from a set of FIFOs using an $E \times E$ crossbar and thus the number of levels of gates

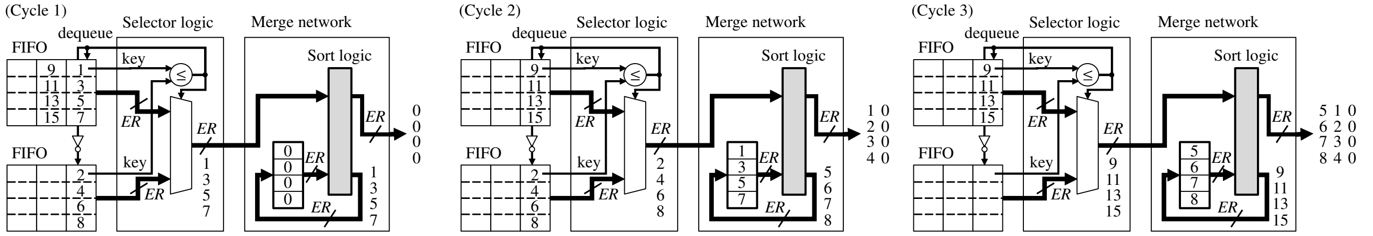

Fig. 2. An example behavior of merging two sorted sequences {1, 3, 5, 7, 9, 11, 13, 15} and {2, 4, 6, 8} using the E-record merge logic of model (b) in Fig. 1. In this example, $E = 4$ and R is the record width.

is increased with increasing E. This results in a significant frequency degradation when E is increased.

Secondly, we describe the behavior of model (b). Previous work [5] adopts this model. It consists of two FIFOs, a selector logic and a merge network. Each FIFO stores a sorted sequence. The merge operation is performed as follows. First, the selector logic compares the two records at the front of two FIFOs and selects the FIFO containing the smaller record. After that, E records of the selected FIFO are input into the merge network. The merge network outputs a sorted sequence at the throughput of E records per cycle. We refer to the merge network with E input/output records per cycle as an *E-record merge network*.

Fig. 2 shows an example behavior of the 4-record merge logic in model (b). Here, the two sorted sequences are $\{1, 3, 5, 7, 9, 11, 13, 15\}$ and $\{2, 4, 6, 8\}$. In previous work [5], [8], an E-record merge network has E registers to store E records to merge correctly. Thus, the merge network in Fig. 2 contains four registers. We call the registers *feedback registers*, the operation of writing records to feedback registers *feed back*, and the record stored in a feedback register a *feedback record*.

In Fig. 2, all four feedback registers must be initialized to the smallest value of all records which is 0 in this example [1], otherwise, the merge logic cannot perform the merge operation correctly. For example, if all feedback registers are initialized to 99, the four records $\{1, 3, 5, 7\}$ are output from the merge network in cycle 1. The merge operation is wrong because 2 is smaller than 3 but the merge network outputs 3 before 2.

In cycle 1, the two records at the front of two FIFOs, 1 and 2, are compared in the selector logic. Since $1 \leq 2$, four records $\{1, 3, 5, 7\}$ in the upper FIFO are input into the merge network. The merge network sorts eight records consisting of four input records and four feedback records in the sort logic, outputs the four smallest records $\{0, 0, 0, 0\}$, and feeds back the four largest records $\{1, 3, 5, 7\}$.

In cycle 2, the two records at the front of two FIFOs, 9 and 2, are compared in the selector logic, and four records $\{2, 4, 6, 8\}$ in the lower FIFO are input into the merge network. Since the four feedback records from the previous cycle are $\{1, 3, 5, 7\}$, the merge network outputs the four smallest records $\{1, 2, 3, 4\}$, and feeds back the four largest records $\{5, 6, 7, 8\}$.

In cycle 3, the operation is performed in the same manner, except the operation of the selector logic. The selector logic selects the upper FIFO because all records in the other FIFO have been dequeued [2]. In this manner, the E-record merge logic merges two sorted sequences.

Model (b) does not require any record reordering between the FIFOs and the merge network. However, the problem is in the merge network. The E-record merge network in [5] is based on Batcher's odd-even merge algorithm [9] and its critical path logic consists of $1 + \log_2 E$ sets of a 2-input multiplexer and a comparator. The critical path logic of another E-record merge network [8] consists of a $2E$-input multiplexer and the accompanying selector logic circuit. Thus, when E is increased, the number of levels of gates in these merge networks increases and their operating frequencies decrease.

There are two ways in order to realize a high-performance hardware merge sorter: (1) to implement the input managers whose number of levels of gates is constant with respect to E in model (a), or (2) to implement the merge network whose number of levels of gates is constant with respect to E in model (b). The first approach may be difficult because eliminating the crossbar switches of the input managers is not simple. Therefore, we focus on model (b) in Fig. 1.

The purpose of this research is to propose a high-performance hardware merge sorter. The key contributions of this paper are as follows.

1) To work correctly, any existing E-record merge logic of model (b) requires E feedback registers in the merge network. We revisit the role and characteristics of the feedback registers, and show that $E - 1$ feedback registers are necessary and sufficient.
2) The merge network architectures for hardware merge sorters proposed in this paper are the first ones where the number of levels of gates is constant when E, the number of records that can be output per cycle, is increased. Our hardware merge sorters achieve significantly higher frequencies compared to existing hardware merge sorters.
3) We implement, synthesize, and evaluate some hardware merge sorters adopting the proposed architectures on a

[1] We actually add a bit to each key to separate the smallest value from valid keys.

[2] To realize the operation, we add a finish bit to each key.

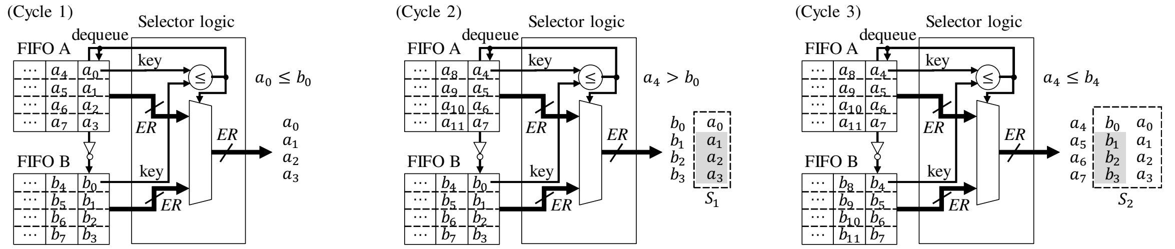

Fig. 3. An example behavior of a part of the E-record merge logic to explain why $E-1$ feedback records are necessary. In this example, $E=4$ and R is the record width.

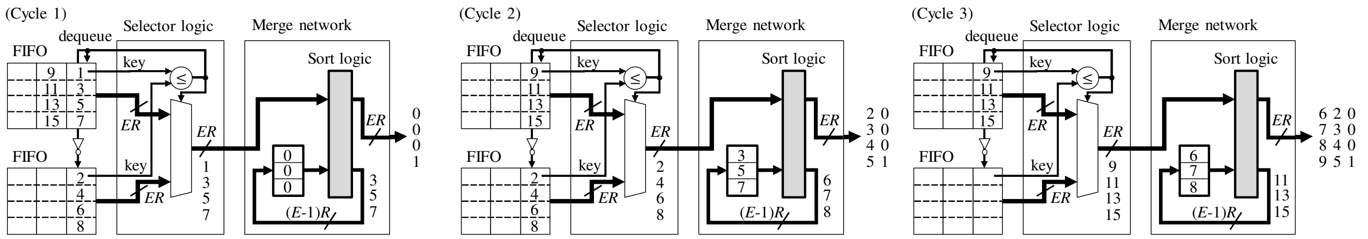

Fig. 4. An example behavior of merging two sorted sequences {1, 3, 5, 7, 9, 11, 13, 15} and {2, 4, 6, 8}, which are the same as the ones in Fig. 2, using a 4-record merge logic which feeds back three records in each cycle. In this example, $E=4$ and R is the record width.

Virtex-7 VX485T FPGA. The evaluation shows that a hardware merge sorter which can output 32 records per cycle operates at 311MHz and achieves 3.13x higher throughput than the state-of-the-art hardware merge sorter [7].

II. Merge network

A. Role and characteristics of feedback registers

In the previous studies [5], [8], the number of feedback registers of an E-record merge logic is E. However, these studies do not provide any explanation of why E feedback registers are necessary. Therefore, before proposing new merge network architectures, we revisit the role and characteristics of feedback registers. We found that $E-1$ feedback registers are necessary and sufficient for correct merge operation.

Fig. 3 shows the example behavior of a part of the E-record merge logic to explain why $E-1$ feedback registers are necessary ($E=4$ in this example). Here, we merge two sequences $\mathbb{A}$ and $\mathbb{B}$ defined in the previous section. We define S_t as the set of all records that are output from the selector logic from cycle 1 to cycle t.

In the initial state, two FIFOs A and B contain all records of sequences $\mathbb{A}$ and $\mathbb{B}$ respectively. In cycle 1, the selector logic compares two head records a_0 and b_0. We suppose that $a_0 \leq b_0$, and thus E records $\{a_0 \sim a_3\}$ are output from the selector logic. Here, S_1 is $\{a_0 \sim a_3\}$.

In cycle 2, the selector logic compares a_4 and b_0 and we suppose that $a_4 > b_0$. So $\{b_0 \sim b_3\}$ are output from the selector logic. Here, S_2 is $\{a_0 \sim a_3, b_0 \sim b_3\}$. Since $a_0 \leq b_0$ as we assumed in cycle 1 and $b_0 \leq b_1 \leq b_2 \leq b_3$, the record a_0 is smaller than or equal to $b_0 \sim b_3$ which are the output records in this cycle. Therefore, **the three records** $a_1 \sim a_3$ highlighted in Fig. 3 (cycle 2) are the only records in S_1 that may be larger than b_0.

In cycle 3, the selector logic compares a_4 and b_4, we suppose that $a_4 \leq b_4$. So $\{a_4 \sim a_7\}$ are output from the selector logic. Since $a_4 > b_0$ as we assumed in cycle 2 and $a_4 \leq a_5 \leq a_6 \leq a_7$, the record b_0 is smaller than $a_4 \sim a_7$ which are the output records in this cycle. It is obvious that $a_0 \sim a_3$ are smaller than of equal to a_4. Therefore, **the three records** $b_1 \sim b_3$ highlighted in Fig. 3 (cycle 3) are the only records in S_2 that may be larger than a_4.

In this example of the 4-record merge logic, in cycle t, there exists a case in which three records in S_{t-1} are larger than the records output from the selector logic and input into the merge network. Thus, three largest records of S_{t-1} must be kept (fed back) in the merge network to be merged with the records output from the selector logic in cycle t. In general, for an E-record merge network, $E-1$ feedback registers are necessary for correct merge operation.

Fig. 4 shows the example behavior of a 4-record merge logic which has three feedback registers. The two sorted sequences are the same as the ones in Fig. 2. In cycle 1, {0, 0, 0, 1} are output from the merge network. The records {2, 3, 4, 5} and {6, 7, 8, 9} are output in cycle 2 and 3 respectively. Note that the output timing of the records is slightly different from the behavior of Fig. 2. We can see that the two sequences are merged correctly. From this example, we can see that $E-1$ feedback registers are sufficient for correct merge operation.

As described before, every existing merge network [5], [8] has E feedback registers to store E feedback records, and thus one feedback register is wasteful.

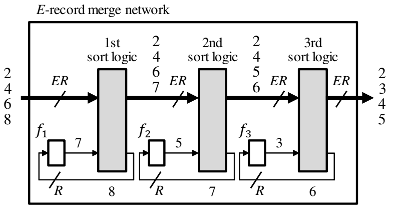

Fig. 5. The basic idea towards our merge networks. In this example, $E = 4$ and R is the record width. The merge network has $E - 1$ sort logics and $E - 1$ feedback registers.

B. Basic idea towards the proposed merge networks

Fig. 5 shows the basic idea towards the proposed merge networks in the case $E = 4$. According to the analysis in section II-A, the merge network has $E - 1$ feedback registers labeled f_i $(1 \leq i \leq E-1)$. The situation of the merge network here is the same as that in cycle 2 in Fig. 4.

The merge network has $E - 1$ sort logics, each of which is a combinational circuit which sorts $E + 1$ records including E sorted records and one feedback record. The sort logics are labeled 1st, 2nd, etc. from the input to the output of the merge network.

The register f_i's output is connected with the input of the i-th sort logic. The largest record among the $E + 1$ output records of the i-th sort logic is fed back to the register f_i. We define that v_{f_i} is the record stored in f_i. Like in the merge network in Fig. 4, these feedback records are set to the smallest value of all records at the initial state. Based on the operation of the sort logics, we have: $v_{f_1} \geq v_{f_2} \geq v_{f_3}$ at any cycle.

We explain the correctness of the merge network depicted in Fig. 5. Let S be the set of seven records consisting of four input records {2, 4, 6, 8} and three feedback records {3, 5, 7}. The 1st sort logic sorts five records {2, 4, 6, 8, 7}, outputs the four smallest records {2, 4, 6, 7} in sorted order, and feeds back the largest record 8 which is also the largest record of S. The 2nd sort logic sorts {2, 4, 6, 7, 5}, outputs {2, 4, 5, 6}, and feeds back 7 which is the second largest record of S. The 3rd sort logic sorts {2, 4, 5, 6, 3}, outputs {2, 3, 4, 5}, and feeds back 6 which is the third largest record of S.

Therefore, the merge network outputs {2, 3, 4, 5} which are the four smallest records of S in sorted order and feeds back {6, 7, 8} which are the three largest records of S to registers f_1, f_2 and f_3 so that $v_{f_1} \geq v_{f_2} \geq v_{f_3}$. The output of this merge network is the same as that in cycle 2 in Fig. 4.

The E-record merge network using $E-1$ sort logics in Fig. 5 outputs the E smallest records in sorted order and feeds back the $E - 1$ largest records every cycle [3]. In this manner, the output record sequence of this merge network is identical to the one of the merge network in Fig. 4 when their input record sequences are the same.

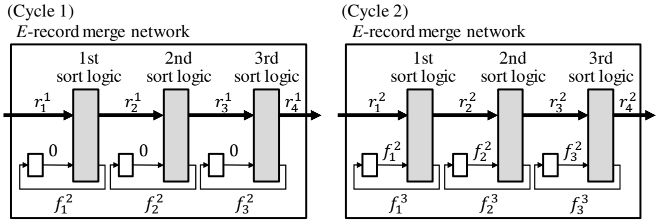

Fig. 6. An example behavior of the merge network in Fig. 5. In this example, $E = 4$.

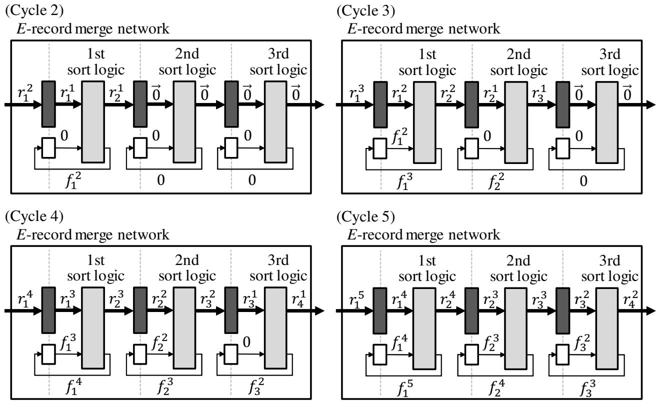

Fig. 7. Our pipelined merge network and its example behavior. In this example, $E = 4$.

C. $(E - 1)$-stage pipelined E-record merge network

We propose two novel merge networks whose number of levels of gates is constant with increasing E, the number of input/output records per cycle. The first one is a pipelined merge network which is based on the merge network in Fig. 5. The second one is described in the next section.

We firstly show an example behavior of the merge network in Fig. 5 from cycle 1 to cycle 2 in Fig. 6. In this example, $E = 4$ and we assume that 0 is the smallest record of all records and all feedback records are initialized to 0 as described in section I. We define r_1^t as the set of E records input into the merge network from the selector logic in cycle t.

In cycle 1, the set of E records input into the merge network is r_1^1. $E+1$ records consisting of r_1^1 and 0, the initial feedback record, are input into the 1st sort logic. Then, the sort logic outputs the E smallest records to the 2nd sort logic and feeds back the largest record. We call the feedback record f_1^2 and the set of the remaining E output records r_2^1. Similarly, the i-th sort logic $(2 \leq i \leq E-1)$ takes as its input $E+1$ records consisting of r_i^1 and 0, outputs the smallest E records (r_{i+1}^1), and feeds back the largest record (f_i^2). The set of E records output by the merge network in this cycle is r_E^1.

In cycle 2, the set of E records input into the merge network is r_1^2. We label the feedback record and the E records output by each sort logic in the same manner as in cycle 1. Note that the feedback records input into the sort logics in this cycle are not 0 but the records fed back in cycle 1. The set of E

[3] This behavior is on the assumptions that E records are provided to the merge network every cycle and the circuit connected to the output of the merge network can always accept outgoing records from the merge network. We will describe the situations where these assumptions are false and our solutions in section II-E.

records output by the merge network in this cycle is r_E^2.

Secondly, we describe our pipelined merge network. Fig. 7 shows this merge network and its example behavior from cycle 2 to cycle 5. The dark gray boxes in this figure are the inserted pipeline registers. They are initialized with the sets of E smallest records which are 0 in our example. These sets of records are labeled $\vec{0}$ in this figure. This is necessary to keep the feedback record of the i-th sort logic 0 until r_i^1 is input into it for correct merge operation.

In cycle 2, $E+1$ records consisting of r_1^1 and 0, the initial feedback record, are input into the 1st sort logic. These input records are the same input records in cycle 1 of the 1st sort logic in Fig. 6. Therefore, the output records are also the same, which are r_2^1 and f_1^2. r_2^1 is stored in a pipeline register and f_1^2 is fed back.

In cycle 3, r_2^1 and 0 are input into the 2nd sort logic. These input records are the same input records in cycle 1 of the 2nd sort logic in Fig. 6. Therefore, the output records are r_3^1 and f_2^2.

In cycle 4, r_3^1 and 0 are input into the 3rd sort logic. These input records are the same input records in cycle 1 of the 3rd sort logic in Fig. 6. Therefore, the output records are r_4^1 and f_3^2. r_4^1 is output from the merge network in this cycle.

As a result, the E output records in cycle 4 of the merge network in Fig. 7 are the same as those in cycle 1 of the merge network in Fig. 6. In the same manner, in Fig. 7, the input records of the 1st sort logic in cycle 3, the 2nd sort logic in cycle 4 and the 3rd sort logic in cycle 5 are the same as those in cycle 2 of the sort logics in Fig. 6. Therefore, the set of E output records in cycle 5 of the merge network in Fig. 7 is r_4^2 which is the same as that in cycle 2 of the merge network in Fig. 6.

It is obvious from the above example that the outputs of two merge networks in Fig. 6 and Fig. 7 are the same with the same input. Thus, the merge network of Fig. 7 also operates correctly.

Fig. 8 shows the block diagram of our proposed merge network with $E-1$ pipeline stages. The sort logic of each pipeline stage consists of E comparators, two 2-input multiplexers, and $E-1$ 3-input multiplexers

Since the sorting operation in a sort logic is done by inserting the feedback record into E sorted records, the multiplexer which outputs the i-th ($2 \leq i \leq E$) smallest record is a 3-input multiplexer and its three inputs include the feedback record and the $(i-1)$-th and i-th smallest records of the E sorted records. The multiplexer which outputs the smallest record is a 2-input multiplexer and its two inputs are the feedback record and the smallest record of the E sorted records. The multiplexer which outputs the largest record is also a 2-input multiplexer and its two inputs are the feedback record and the largest record of the E sorted records.

Therefore, E affects the numbers of pipeline stages, multiplexers and comparators. E does not affect the number of levels of gates.

This proposed merge network has two major advantages. First, the critical path logic is simple because it includes only one comparator of two keys and one 3-input multiplexer. This results in a significant frequency improvement compared with existing merge networks. The other advantage is that the number of levels of gates is constant with increasing E.

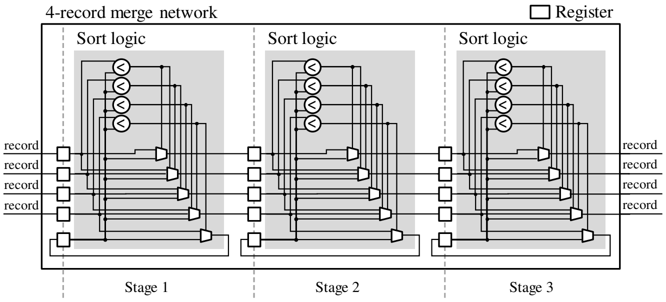

Fig. 8. The proposed E-record merge network architecture with $E-1$ pipeline stages where the number of levels of gates is constant with increasing E.

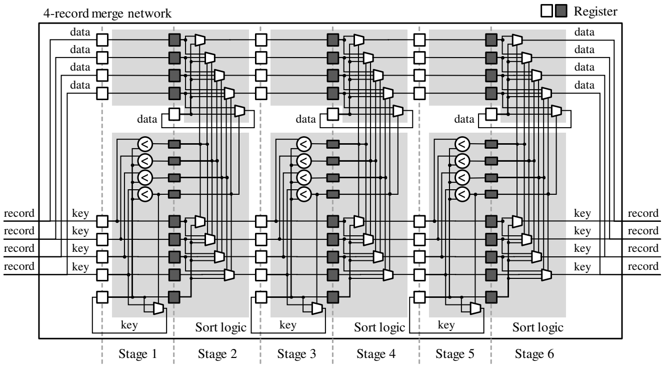

Fig. 9. The proposed more deeply pipelined E-record merge network architecture: $2(E-1)$ pipeline stages.

D. $2(E-1)$-stage pipelined E-record merge network

To improve the frequency of the merge network in Fig. 8, we propose a more deeply pipelined merge network.

Fig. 9 shows our $2(E-1)$-stage pipelined E-record merge network. The key and the data of each record are processed in separate datapaths. The idea here is to divide each sort logic in Fig. 8 into two pipeline stages as below.

- The first stage compares the key of each input record with the key of the feedback record and feeds back the largest key.
- The second stage selects the output records of the sort logic and feeds back the data of the record whose key is the largest key determined in the previous pipeline stage.

The dark gray boxes in Fig. 9 are the additional pipeline registers. To maintain the throughput of E records per cycle, the sort logic feeds back the largest one among all keys input to the sort logic.

The critical path logic of the first pipeline stage consists of one comparator of two keys and one 2-input multiplexer of keys. The critical path logic of the second pipeline stage consists of only one 3-input multiplexer of keys or data.

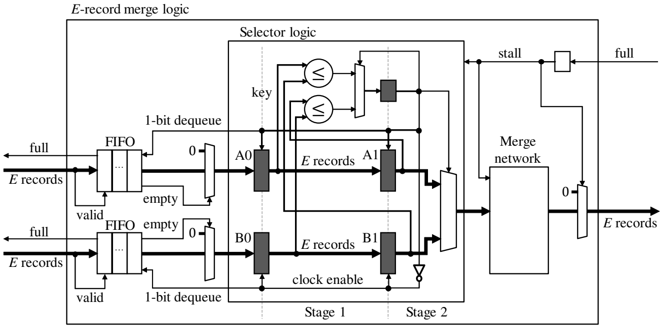

Fig. 10. The proposed merge logic architecture.

Since the critical path delay of these two stage logics are shorter than that of the sort logic in Fig. 8, a higher frequency can be achieved. The number of pipeline stages of the merge network in Fig. 9 is increased to $2(E-1)$ from $E-1$ stages in Fig. 8.

Our evaluation shows that the merge network in Fig. 9 is faster than the one in Fig. 8. Therefore, we use the merge network in Fig. 9 as the proposed merge network in the evaluations below.

E. Modifications for actual implementations

So far, we have explained the E-record merge logic and the E-record merge networks in the assumption that each FIFO never becomes empty until all records of the sequence are dequeued. However, in actual implementations, FIFOs may become empty temporarily.

In addition, we have assumed so far that the merge logic can always output E records. However, in actual implementations, the merge logic sometimes must stop the record output because the logic connected with the output of the merge logic is not ready to accept records from the merge logic.

We deal with the above problems by adding some functionalities to the merge logic. When any FIFO is empty, we insert dummy records which have the smallest key into the merge network. Dummy records are discarded after being output from the merge network. By this way, the feedback registers are not updated by dummy records and the merge logic performs merge operation correctly. When the merge logic must stop the record output, we stall all registers in the merge network and some registers in the merge logic.

III. Merge logic

Fig. 10 shows the block diagram of our merge logic. The multiplexer between a FIFO and the selector logic is used to insert dummy records when the FIFO is empty as described in section II-E. When the input full signal is set, the stall signal is set. The stall signal stalls the merge network and the selector logic.

To fully exploit the performance gained by using the proposed merge network, other logics in the merge logic should not be a critical path logic. We experimentally found that the critical path delay of the naive selector logic shown in model (b) in Fig. 1 becomes longer than that of the proposed merge network when E increases. To solve the problem, we propose a pipelined selector logic.

The naive selector logic shown in model (b) in Fig. 1 compares two records at the front of two FIFOs, selects the FIFO containing the smaller record and outputs the E records of the selected FIFO to the merge network.

In our merge logic, records dequeued from the FIFOs are latched into two stages of pipeline registers ($A0, A1, B0, B1$ in Fig. 10) before being sent to the merge network. Let $X^t_{A0}, X^t_{A1}, X^t_{B0}$, and X^t_{B1} be the sets of E records stored in $A0, A1, B0$, and $B1$, respectively, at cycle t. Let $m^t_{A0}, m^t_{A1}, m^t_{B0}$, and m^t_{B1} be the smallest records of $X^t_{A0}, X^t_{A1}, X^t_{B0}$, and X^t_{B1}, respectively.

At cycle t, the selector logic outputs X^t_{A1} if $m^t_{A1} \leq m^t_{B1}$ or X^t_{B1} otherwise. The comparison of m^t_{A1} and m^t_{B1} is performed at cycle $t-1$ and the result is latched into a pipeline register to be used at cycle t. If X^{t-1}_{A1} is output at cycle $t-1$, we will have m^t_{A1} be m^{t-1}_{A0}, and m^t_{B1} be m^{t-1}_{B1}. On the other hand, if X^{t-1}_{B1} is output at cycle $t-1$, we will have m^t_{A1} be m^{t-1}_{A1}, and m^t_{B1} be m^{t-1}_{B0}. Thus, the selector logic [4] contains two comparators and one multiplexer to select one of the comparison results as shown in Fig. 10.

The critical path logic of the proposed selector logic contains one comparator of two keys and one multiplexer of two 1-bit select signals, or only one multiplexer of two sets of records, each with E records. On the other hand, the critical path logic of the naive selector logic in model (b) in Fig. 1 consists of one comparator of two keys and one multiplexer of two sets of records, each with E records. Therefore the critical path delay of the proposed selector logic is much shorter.

IV. Evaluation

A. Hardware merge sorter

N records can be sorted by recursively using a merge logic. When the merge logic outputs E records per cycle, the total sorting time is roughly estimated to be $\frac{N}{E}\log_2 N$ cycles [7].

We define W as the number of sorted sequences which a hardware merge sorter can merge simultaneously. The sorting time can be reduced by a factor of $\log_2 W$ by simultaneously merging W sorted sequences using a hardware merge sorter called a merge tree which is constructed from multiple merge logics. Therefore, most recent hardware merge sorters are based on merge trees [5]–[7], [10].

In this paper, we use the state-of-the-art merge tree proposed in [7] to evaluate our proposed merge logic architecture. As shown in Fig. 11, this merge tree is constructed from merge logics with different sizes and couplers, each is responsible for concatenating two input record sets into one output record set. The number of input sequences W is equal to E, the number of records can be output per cycle by the merge logic at the

[4] At any cycle t, the selector logic explained here always outputs X^t_{A1} if $m^t_{A1} = m^t_{B1}$. Because this unevenness may degrade the throughput of the hardware merge sorter discussed in section IV-A, we develop a mechanism where the selector logic can alternately output X^t_{A1} and X^t_{B1} when $m^t_{A1} = m^t_{B1}$. However, for simplicity, this mechanism is not shown in Fig. 10.

TABLE I
EVALUATION RESULTS OF OUR E-RECORD MERGE TREE IN COMPARISON WITH THOSE OF THE E-RECORD MERGE TREE PROPOSED IN [7].

E	SHMS (Proposal)					PMT [7]				
	Freq. [MHz]	Register	LUT	Active rate	Throughput [Gb/s]	Freq. [MHz]	Register	LUT	Active rate	Throughput [Gb/s]
2	440.5	1,316 (0.22%)	512 (0.17%)	0.984	55.4	248.5	328	853	0.983	31.3
4	421.8	6,417 (1.06%)	2,668 (0.88%)	0.975	105.2	213.1	1,534	4,278	0.978	53.4
8	397.0	24,669 (4.06%)	11,150 (3.67%)	0.972	197.5	163.3	5,287	16,016	0.974	81.5
16	343.5	90,141 (14.85%)	44,558 (14.68%)	0.969	340.8	133.4	16,299	47,001	0.973	132.9
32	311.7	330,148 (54.37%)	175,936 (57.95%)	**0.968**	**617.9**	99.2	45,445	142,179	0.970	**197.1**

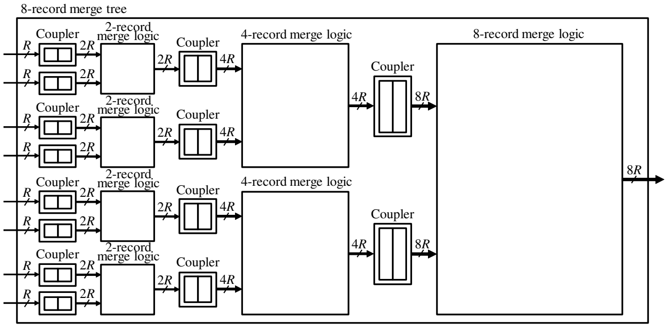

Fig. 11. The state-of-the-art E-record merge tree [7] used in our evaluation. In this example, $E = 8$ and R is the record width.

root. We call the merge tree that has an E-record merge logic at its root an *E-record merge tree*.

The effective throughput of an E-record merge tree becomes smaller than E records per cycle if the dequeue process of one of the input sequences stalls, especially when the input data are not randomly and uniformly distributed, e.g. some input sequences contain many small records while some others contain many large records.

However, the authors of [7] showed that the throughput degradation caused by the above problem is negligible when the input data are randomly and uniformly distributed among the input sequences, which can be achieved by randomly reshuffling the data set before sending it to the merge tree.

B. Evaluation setup

To evaluate our merge tree, we use four metrics: resource utilization, active rate, operating frequency and throughput.

We define the active rate α ($0 < \alpha \leq 1$) of a merge tree as the ratio the number of cycles when valid records are output from the merge tree divided by the number of elapsed cycles. α depends on both the distribution of input data and the sizes of the FIFOs of the merge logics [7].

We use the same methodology as [7] to implement the FIFOs effectively: we use shift register primitive (SRL16E) and the size of each FIFO is set to 16 because one SRL16E can be used as a 16-bit shift register. This configuration makes the comparison between our merge tree and the one in [7] fair.

The throughput T of an E-record merge tree is calculated by the following equation: $T = \alpha \times F \times E$, where F is the operating frequency.

In all evaluations, the width of each record is 66-bit, where the data field is 32-bit and the key field is 34-bit. Two most significant bits of the key field are the finish bit and the valid bit as explained in section I. Since these two bits are just for the control of our merge tree, we calculate the throughput of our merge tree by regarding the record width as 64-bit for the fair comparison with [7] where the key field and the data field of each record are 32 bits wide.

Feedback registers are initialized with dummy records of which the valid bit is zero. After all records in a sequence have been input to the merge tree, the finish bit is set for this sequence. Since a record with the finish bit set is larger than all valid records, the selector logics in the merge tree always prioritize valid records, and thus the merge tree can merge all sequences correctly.

The target FPGA is Virtex7 XC7VX485T which is the same one used in [7]. Our implementations are entirely written in Verilog HDL; only the FIFOs are directly instantiated using the FPGA hardware primitives. We do not manually floorplan the designs. They are placed and routed automatically by Vivado 2016.3.

C. Evaluation results

The main evaluation results are summarized in Table I. We compare the proposed hardware merge sorter (SHMS) with the state-of-the-art merge tree (Parallel Merge Tree, PMT) proposed in [7].

1) Resource utilization analysis: Table I shows that the resource utilization of SHMS is more than that of PMT. When $E = 32$, the register and the LUT utilization of SHMS are 7.26x and 1.24x of PMT. SHMS achieves significantly higher frequency and throughput than PMT thanks to the constant number of levels of gates with increasing E as we will show below, but SHMS requires more resources than PMT.

2) Active rate analysis: To examine the active rate of SHMS, we use Synopsys VCS simulator. As same as [7], the input data sequences of SHMS are randomly generated. Each of them is 1M records long.

We can see in Table I that both SHMS and PMT exhibit very high active rates in every case and the lowest active rate is still 0.968. Thus the effect of the active rate on throughput is small.

3) Frequency analysis: Table I shows the frequencies of SHMS and PMT in five cases: E = 2, 4, 8, 16, and 32. The result shows that we have achieved significantly higher

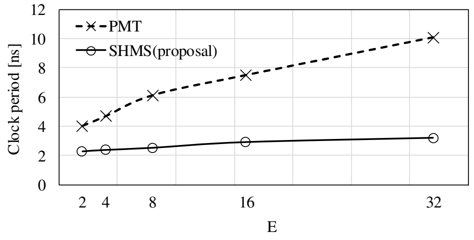

Fig. 12. Clock periods in five cases: $E = 2$, 4, 8, 16, and 32.

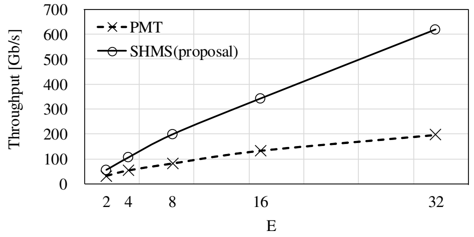

Fig. 13. Throughputs in five cases: $E = 2$, 4, 8, 16, and 32.

frequencies. This is because the critical path logic of SHMS is simple. When $E = 32$, SHMS is 3.14x faster than PMT.

The number of levels of gates of SHMS is constant with respect to E. Therefore, the increase of the clock period when increasing E is caused by the increase of the complexity of the placement and routing task.

Fig. 12 shows the clock periods of SHMS and PMT. We can see that, when E increases, the clock period of SHMS increases slightly whereas the clock period of PMT increases significantly. The clock period of SHMS increases only about 41% when E increases from 2 to 32. This is significantly better than PMT whose clock period increases about 150% when increasing E from 2 to 32.

4) Throughput analysis: According to the previous analysis, the effect of the active rate on throughput is small and SHMS exhibits very high operating frequencies. Therefore, SHMS can achieve a very high throughput.

Fig. 13 shows the throughputs of SHMS and PMT in five cases: $E = 2$, 4, 8, 16, and 32. Thanks to the constant number of levels of gates with respect to E, the throughput of SHMS scales slightly less than linear with E. On the other hand, the improvement rate in throughput of PMT is limited to about 6.3x when increasing E from 2 to 32. As a result, SHMS achieves very high throughput of up to 617.9 Gb/s, which is 3.13x higher than the highest throughput of PMT.

D. Discussion

Firstly, we discuss the sorting time of hardware merge sorters. According to section IV-A, the sorting time of a hardware merge sorter is roughly estimated to be $\frac{N \log_W N}{T}$ seconds, where N is the number of records to be sorted, W is the number of sequences and T is the throughput (records per second) of the sorter. From this estimation, we can see that the sorting time of a hardware merge sorter can be reduced by increasing W and improving T in general. Although some studies focus on the former approach, this study and PMT focus on the latter one.

Both PMT and SHMS achieve their shortest sorting time when $E = 32$ and $W = 32$. On these same parameters, since SHMS provides 3.13x higher throughput than PMT as described in section IV-C, we say that SHMS is 3.13x faster than PMT in terms of sorting time.

Secondly, we mention the superiority of the deeply pipelined architecture in section II-D compared to the one in section II-C. Our experimental results show that, if we use the architecture in section II-C instead of the deeply pipelined one for implementing SHMS, the operating frequency of SHMS decreases significantly. For example, in the case of $E = 32$, the frequency is reduced from 311MHz to 220MHz.

Finally, we mention the impact of the pipelined selector logic in Fig. 10 on the operating frequency of SHMS. Our experimental results show that, replacing the pipelined selector logic with the naive one shown in model (b) in Fig. 1 results in a significant decrease in the operating frequency of SHMS. For instance, in the case of $E = 32$, the frequency is reduced to 263MHz from 311MHz.

V. Conclusion

In this paper, we proposed a novel and high-performance hardware merge sorter. The proposed hardware merge sorter is the first one where the number of levels of gates is constant with increasing the number of records output per cycle. Our evaluation shows that the proposed hardware merge sorter which simultaneously merges 32 sorted sequences and can output 32 records per cycle operates at 311MHz and achieves the throughput of 617.9 Gb/s which is 3.13x higher than the state-of-the-art hardware merge sorter.

References

[1] H. Inoue and K. Taura, “SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures,” *Proc. VLDB Endow.*, vol. 8, no. 11, 2015.

[2] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, “PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort,” *Proc. VLDB Endow.*, vol. 8, no. 12, 2015.

[3] D. Merrill and A. Grimshaw, “High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing,” *Parallel Processing Letters*, vol. 21, no. 02, 2011.

[4] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens, “Efficient Parallel Merge Sort for Fixed and Variable Length Keys,” in *InPar*, 2012.

[5] J. Casper and K. Olukotun, “Hardware Acceleration of Database Operations,” in *FPGA*, 2014.

[6] D. Koch and J. Torresen, “FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on Fpgas for Large Problem Sorting,” in *FPGA*, 2011.

[7] W. Song, D. Koch, M. Luján, and J. Garside, “Parallel Hardware Merge Sorter,” in *FCCM*, 2016.

[8] S. Mashimo, T. V. Chu, and K. Kenji, “Cost-Effective and High-Throughput Merge Network Architecture for the Fastest FPGA Sorting Accelerator,” in *HEART*, 2016.

[9] D.-L. Lee and K. E. Batcher, “A Multiway Merge Sorting Network,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, 1995.

[10] G.-S. Liu and H.-H. Chen, “Parallel Merge Module for Combining Sorted Lists,” *IEE Proceedings E - Computers and Digital Techniques*, vol. 136, no. 3, 1989.