Parallel Hardware Merge Sorter

Wei Song*†, Dirk Koch†, Mikel Luján†, and Jim Garside†
*Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, UK
†School of Computer Science, University of Manchester, Manchester M13 9PL, UK
ws327@cam.ac.uk; {dirk.koch; mikel.lujan; james.garside}@manchester.ac.uk

***Abstract*—Sorting has tremendous usage in the applications that handle massive amount of data. Existing techniques accelerate sorting using multiprocessors or GPGPUs where a data set is partitioned into disjunctive subsets to allow multiple sorting threads working in parallel. Hardware sorters implemented in FPGAs have the potential of providing high-speed and low-energy solutions but the partition algorithms used in software systems are so data dependent that they cannot be easily adopted. The speed of most current sequential sorters still hangs around 1 number/cycle. Recently a new hardware merge sorter broke this speed limit by merging a large number of sorted sequences at a speed proportional to the number of sequences. This paper significantly improves its area and speed scalability by allowing stalls and variable sorting rate. A 32-port parallel merge-tree that merges 32 sequences is implemented in a Virtex-7 FPGA. It merges sequences at an average rate of 31.05 number/cycle and reduces the total sorting time by 160 times compared with traditional sequential sorters.**

***Index Terms*—Merge sort, sorting network, parallel sorting, FPGA.**

I. Introduction

As a key part in numerous computer algorithms, sorting has tremendous usage in many applications. It is an important step of the reduce function used in the MapReduce programming model [1], which is the *de facto* distributed model for processing massive data sets on server clusters. For relational databases, join is one of the fundamental query operations. As one of the most utilized join implementation, sort-merge join [2] combines two tables by firstly sorting their keys. Low latency sorting is also required in some scientific computing [3]. In fact, sorting is so important that various sorting benchmarks [4] have been defined to regularly rank the top sorters ever built.

The methods of accelerating sorters have been researched for several decades using various devices, such as parallel computers [5], dedicated application specific integrated circuits (ASICs) [6]–[8], field programmable gate arrays (FPGAs) [9]–[12], multiprocessors [13]–[15] and recently general-purpose graphic processing units (GPGPUs) [16], [17]. Sorting becomes increasingly important these days due to the fast growing amount of data in certain applications, where database is one of the most active areas [11], [12].

The state-of-the-art research on sorter accelerating concentrates on software sorting algorithms [4]. A large data set can be divided into disjunctive subsets [15]; therefore, subsets can be sorted by multiple threads in parallel. Since this partition procedure is highly data dependent, it is not directly applicable for hardware sorters, although hardware sorters have the potential of providing high-speed and low-energy sorting solutions. Up to now, there is no easy way of making hardware sorters run in parallel.

Dr Luján is funded by a Royal Society University Research Fellowship.

Existing hardware sorters are either parallel or sequential sorters. Parallel sorters, such as sorting networks [18], can sort up to only hundreds of numbers [8] due to the limited number of I/O ports. As an alternative, sequential sorters [19] can sort a large data set but at a low sorting rate of usually 1 number/cycle. Such low speed significantly limits the benefit of using FPGAs as sorting accelerators. Without a method to sort a large data set in parallel, hardware sorters are actually slower than software sorters running on processors considering the gap in clock frequencies.

This paper proposes a parallel merge-tree which merges multiple sorted data sequences at a speed proportional to the number of sequences. It has three major advantages over sequential sorters:

- The sorting rate is significantly increased from 1 number/cycle up to 1 number/cycle/sequence.
- Merging more than 2 sequences significantly reduces the number of passes (runs) to sort a large data set.
- The parallel merge-tree merges data in a streamed fashion. The size of the sorted data is not constrained by the available on-FPGA memory, which is used only as communication buffers for the main memory.

A 32-port parallel merge-tree is implemented in a Xilinx Virtex-7 XC7VX485T FPGA [20]. It merges 32 sequences at a rate up to 32 number/cycle. When sorting a large data set, using this 32-port parallel merge-tree reduces the total sorting time by nearly 160 times compared with sequential sorters.

II. Existing Sorters

The proposed parallel merge-tree is developed from existing sorters and can be used along with other designs to form large sorting systems. The related hardware and software sorters are briefly reviewed to provide a limited background introduction. As a sorter can sort numbers in either descending or ascending order, all sorters described in this paper use the descending order to avoid ambiguity.

A large scale sorting system is normally composed of tens to thousands of server nodes [4]. The input data are divided into disjunctive subsets using foreknown statistics or an initial sampling [21]–[23]. These subsets are sorted by parallel servers simultaneously and then merged into a final sorted sequence.

Subset partition and workload balancing among servers are fulfilled by software [21], [23] due to their data dependence and irregular control patterns. The actual sorting acceleration happens in the individual sorting server which sorts a subset or merges the final sequence. Current servers use off-the-shelf solutions such as high performance multiprocessors [21]–[23] or GPGPUs [16], [17]. The proposed parallel merge-tree can be dynamically reconfigured to an FPGA as a sorting accelerator cooperating with a server, which is potentially more energy and cost efficient than sorting in processors. Although this configuration is rarely used in present datacenters, it is likely to become popular with the availability of high-performance embedded cores in future large capacity FPGAs (e.g. Xilinx UltraScale+ or Altera Stratix 10).

Servers choose different sorting algorithms according to statistics and available resources. To reduce sorting time, it is important to sort in parallel but not all sorting algorithms can be parallelized easily. *Merge sort* and *radix sort* are two extensively utilized algorithms which are both parallelizable but with very different nature. Merge sort is a comparison sorting algorithm which merges normally two sorted sequences into one. An unsorted data set can be sorted by recursively applying merge sort in multiple passes (also called runs in related literature). The single-thread time complexity of sorting N numbers is $O(N \log N)$. Radix sort is a distribution sorting algorithm which sorts numbers by processing individual digits. The sorting time is therefore proportional to the bit-width of numbers. The single-thread time complexity of sorting N numbers of k bits is $O(N \cdot k)$. Generally speaking, merge sort is faster than radix sort when sorting a bounded size of wide numbers ($k > \log N$), which could be the case for a server that sorts only a subset of the whole data. Merge sort is the target sorting algorithm researched in this paper.

Merge sort can be parallelized for running on multi-thread processors [15], [17]. The two input sequences are partitioned into an arbitrary number of disjunctive and equal-sized segments in linear time [15]. Then each segment is merged on a different thread. For a server with p hardware threads, the time complexity of a parallel merger is $O(N/p \cdot \log N + \log p \cdot \log N)$ [15], providing a speed up slightly less than p.

Hardware sorters are circuit implementations of certain sorting algorithms using ASICs or FPGAs. Most hardware sorters can be classified into either parallel sorters or sequential sorters. The most used parallel sorters are Bitonic and odd-even sorting networks [18]. Both of them sort N numbers in $O(\log^2 N)$ cycles. However, the size of a sortable sequence is constricted by the number of I/O ports, which is limited to hundreds [8] in current VLSI technology. Sequential sorters are the actual sorters able to sort large scale data sets.

Most sequential sorters implement single-thread sorting algorithms in hardware and run in a streamed fashion. The sorted data is normally produced at a constant rate of 1 number/cycle, such as the parallel shift sort (insertion sort) [6], the up/down sorter (heap sort) [24] and the FPGAsort (merge sort) [10]. In all sequential sorters, FPGAsort is currently the most area efficient sorter thanks to its smart use of on-chip memory. The size of a sortable sequence is usually constricted by the accessible storage space (on-chip and off-chip), which can be huge. Compared with hardware parallel sorters, sequential sorters can sort a large scale data set but at a very low speed.

Numerous attempts have been made to parallelize sequential sorters. For hardware merge sorters, both the early parallel merge module [25] and the recent high bandwidth sort merge unit [12] use a tree of merge units to merge more than 2 input sequences simultaneously. However, both sorters choose to enforce non-stall flow control for a constant sorting rate. As a result, all merge units in the tree must run at the maximum rate, which significantly increases area and the required memory bandwidth. Such sorters are not very scalable. The more recent high bandwidth sort merge unit is able to merge up to 8 sequences but requires a staggering memory bandwidth of more than 6 times of the sorting rate [12].

The parallel merge-tree proposed in this paper also uses a tree of merge units. The decisive difference with the aforementioned two is to allow stalls and variable sorting rate in order to pursue a high accumulated sorting speed. In this way, non-root merge units run at lower sorting rates than the root one and consume less area. More importantly, the required memory bandwidth is reduced to just the maximum sorting rate. Compared with the software parallel merger [15], merging more than 2 sequences in each pass provides extra speed up due to the reduced number of passes.

III. Multirate Merging

A traditional hardware merge sorter, such as the FPGA-sort [10], produces sorted data at a speed of 1 number/cycle. A multirate merger (MM) is able to merge two sorted sequences at a speed much faster than 1 number/cycle.

A. Comparator units

Various comparator units [8] perform the basic comparing and exchanging operations needed by the sorters in this paper. A full compare-and-exchange (CAE) unit compares 2 input numbers and outputs both of them in order. Fig. 1a depicts the implementation (left) of a CAE unit and its symbol (right). When only 1 output is needed (assuming the larger one), a CAE unit is reduced to a compare-and-select (CAS) unit [8] as shown in Fig. 1b. For some sorters, the selection result of a CAS unit is used by further circuitry. Such a CAS unit with a selection feedback signal s is depicted in Fig. 1c.

B. Parallelization

The structure of a FIFO merge sorter (unoptimized FPGA-Sort) [10], [26] is illustrated in Fig. 2. It comprises three parts: two FIFOs to store the input sequences and a CAS unit to output the largest number remaining in the two FIFOs.

A multirate merger is an expanded FIFO merge sorter which shares the same overall structure but with more complicated components in each part. An abstract view of a multirate merger which provides P sorted number per cycle, MM(P), is depicted in Fig. 3. It also needs two FIFOs to store the input sequences but the CAS unit is replaced with a parallel sorter

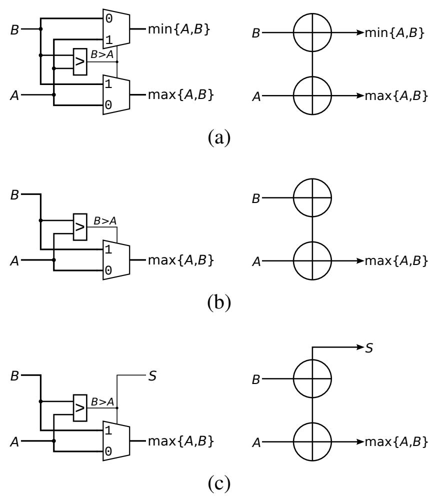


Fig. 1. Implementations and symbols of comparator units: (a) Compare-and-exchange (CAE), (b) compare-and-select (CAS), and (c) CAS with a selection feedback.

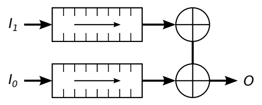


Fig. 2. A FIFO merge sorter.

which extracts and sorts the largest P numbers remaining in the two FIFOs. Since these numbers may come from the same FIFO due to imbalanced distribution, the parallel sorter needs to pick them from a total of $2P$ numbers.

For the multirate merger to work, two issues must be resolved:

- The unchosen numbers must be retained in FIFOs for the next cycle. Therefore, the selection of the largest P numbers must be finished in one cycle to avoid extra feedback delay.
- The FIFOs have a variable data rate between 0 to P number/cycle, which is not directly supported by normal FIFO designs.

C. Single-cycle selection

The solution for the single cycle selection is to use a Bitonic partial merger, which is a part of a Bitonic sorting network and has been analysed recently by Farmahini-Farahani [8].

An 8-to-4 Bitonic partial merger is shown in Fig. 4. It reads numbers from two sorted sequences $[I_3 \cdots I_0]$ and $[I_7 \cdots I_4]$, picks the largest 4 numbers, and shuffles them into a sorted sequence $[O_3 \cdots O_0]$. The merger is fully pipelined into 3 stages and the selection process occurs on the first stage.

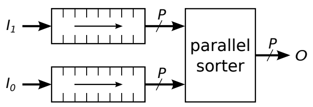


Fig. 3. MM(P), a P-port multirate merger (abstract view).

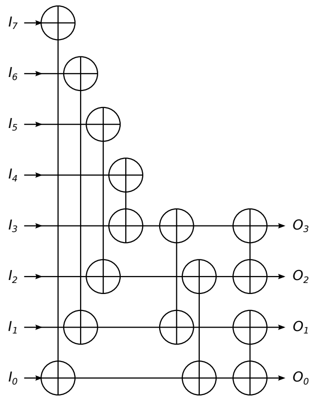


Fig. 4. An 8-to-4 Bitonic partial merger.

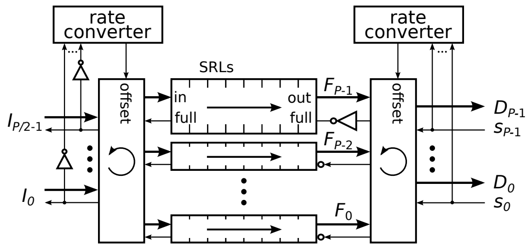


Fig. 5. An FIFO for multirate mergers.

For an MM(P), a $2P$-to-P Bitonic partial merger is used to choose the largest P numbers remaining in the two FIFOs and sort them in a pipelined way. The selection result is available from the first pipeline stage; therefore, it can be used to retain the unselected data in the same cycle. Furthermore, Bitonic sorting networks are known to be area efficient. Since the Bitonic partial merger is a part of the sorting network, it is also area efficient and scalable.

D. Area-efficient FIFO implementation

The FIFOs are shallow but have a variable output rate from 0 to P number/cycle depending on the selection result. It is not easy to use dual-port memory blocks due to this variable rate and a full register-based design is area consuming. The most area-efficient solution so far is to divide a wide FIFO into P parallel narrow FIFOs, each of which is only 1 number wide. Since now the output rate of each narrow FIFO is 0 or 1 number/cycle, it can be implemented in special shift register look-up tables (SRLs) which are abundant in Virtex FPGAs [27]. Every SRL can be used as a 16-bit shift register.

The detailed implementation is demonstrated in Fig. 5. It has P narrow FIFOs implemented in SRLs. Two crossbars connect FIFOs to the input sequence (I) and the Bitonic partial merger (D). The selection result, denoted as s, is fed back to individual FIFOs to identify whether there is shift occurring in the current cycle. Also under the control of the selection result, the output crossbar ensures the FIFO that has the largest remaining number is always connected to D_0.

The output crossbar is implemented as an array of barrel shifters [28] controlled by signal *offset*. Depending on the

TABLE I
CONVERTING s TO *rate*

$[s_{P-1} \cdots s_0]$	*rate*
$\forall i, s_i = 0$	0
$(\forall i < m, s_i = 1) \wedge (\forall i \geq m, s_i = 0)$	m
$\forall i, s_i = 1$	P

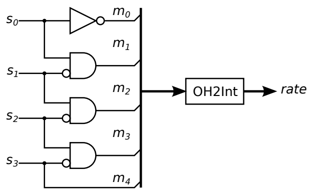


Fig. 6. A rate converter for MM(4).

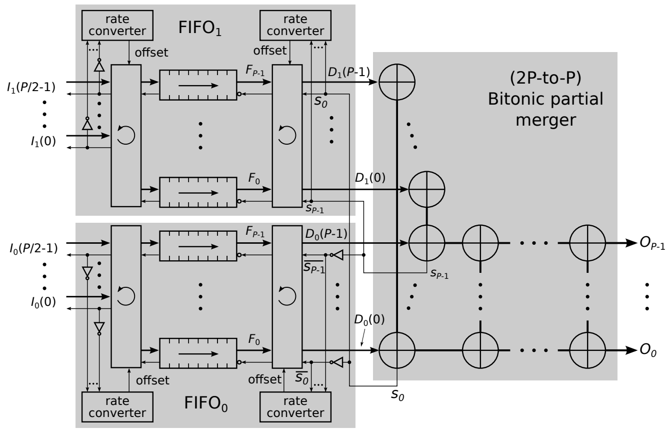


Fig. 7. MM(P), a P-port multirate merger (implementation).

value of *offset*:

$$[D_{P-1} \cdots D_0] = [F_{\text{offset}-1} \cdots F_0, F_{P-1} \cdots F_{\text{offset}}] \quad (1)$$

The input sequence is sorted in descending order. Initially, the largest number is stored in F_0 and connected to D_0 (*offset* = 0). To ensure that D_0 always has the largest number and D is in descending order, *offset* tracks the selection result closely:

$$\textit{offset'} = \textit{offset} + \textit{rate} \quad (2)$$

where *offset'* is the next cycle's value of *offset* and *rate* is the amount of numbers shifted in the current cycle, accumulated from the selection result s:

$$\textit{rate} = \sum_{i=0}^{P-1} s_i \quad (3)$$

Fortunately the conversion from s to *rate* does not really need an accumulator since D is sorted. For any $D_i \in D$, if D_i is selected, $[s_i \cdots s_0]$ must be all ones because any number in $[D_{i-1} \cdots D_0]$ is larger than D_i and must be selected as well. As a result, the conversion from s to *rate* can be summarized in Table I. The corresponding implementation of a rate converter is depicted in Fig. 6. The first column of AND/INV gates translate s into m which is the one-hot format of *rate*. Then a one-hot to binary (OH2Int) circuit provides the *rate*.

The FIFO inputs work in a similar way with outputs. The data rate of the input sequence is $P/2$ number/cycle. The input crossbar (another array of barrel shifters)[1] ensures the input sequence is stored in the correct narrow FIFOs in a round-robin style. The *offset* signal for the input crossbar is controlled by the *full* signal of the FIFO inputs.

E. Multirate merger

Putting all components together, Fig. 7 reveals the implementation of an MM(P). Two sorted input sequences (I_0 and I_1) are stored in two parallel FIFOs. The outputs of FIFOs, denoting the largest P numbers of each FIFO, are fed to a $2P$-to-P Bitonic partial merger, which selects the largest half and sorts them into the output sequence O at a rate of P number/cycle.

The selection result (s) is fed to the two FIFOs slightly differently. For FIFO$_0$, the selection result is inverted:

$$s_{\text{FIFO}_0}(i) = \bar{s_i} \quad (4)$$

When s_i is 0, $D_0(i)$ is selected rather than $D_1(P-i)$, therefore $s_{\text{FIFO}_0}(i)$ should be 1. As for FIFO$_1$, the selection result is flipped due to the flipped connection to the Bitonic merger:

$$s_{\text{FIFO}_1}(i) = s_{P-i} \quad (5)$$

When s_{P-i} is 1, $D_1(i)$ is selected rather than $D_0(P-i)$, therefore $s_{\text{FIFO}_1}(i)$ should be 1.

The critical path of a multirate merger starts from the FIFO outputs, traverses through the first stage of the Bitonic merger, the selection feedback and finally to the full signal of the FIFO outputs. Since both the FIFO and the Bitonic merger is fully parallelized, the output crossbar is the only component affected by the number of ports[2]. Normally the latency of a crossbar is roughly linear with $\log P$. The speed scalability of multirate mergers is expected to be $\log P$ as well.

The area of a multirate merger is dominated by the narrow FIFOs and the Bitonic partial merger. The area of FIFOs is proportional to P while the area of the Bitonic partial merger is linear with $P \log P$ [8]. As a result, the area complexity of an MM(P) is $O(P \log P)$.

F. Rate mismatch

The FIFO input data rate is set to $P/2$ number/cycle for an MM(P). When numbers are perfectly balanced between sequences, the output rate is P number/cycle. The $P/2$ number/cycle input rate is enough to maintain the output rate. When numbers are not balanced, the Bitonic partial merger may consecutively choose numbers from one sequence at a rate larger than $P/2$ number/cycle, which causes rate mismatch in

[1]A crossbar is needed when the length of input sequences is not always times of $P/2$; otherwise a de-multiplexer is enough.

[2]For simplicity, we ignore the timing effect caused by increased area for this analysis.

both FIFOs. If one FIFO is eventually empty, the multirate merger is stalled waiting the empty FIFO to be refilled.

There are two ways to handle this issue: One is to increase the input rate to P number/cycle in order to avoid stalls, the other one is to accept stalls with flexible control logic. Rather than increasing the input rate as the high bandwidth sort merge unit [12] does, the multirate merger adopts the latter solution because the accummulated input rate of the former solution is unsustainable when merging mutiple sequences. More insights will be provided in Section IV-B where the parallel merge-tree has been introduced.

G. *Optimization for skewed data*

When a data set is skewed, the probability of duplicated numbers increases. Since CAS units (shown in Fig. 1) choose a fixed input when the two inputs are equal, duplicated numbers cause stalls in a way similar to the imbalanced data distribution.

An easy way to resolve this issue is to ask the CAS units in the first stage of the Bitonic partial merger to choose different inputs. The rule is simple: For CAS_i that compares $D_0(i)$ and $D_1(P-i)$, when the two numbers are equal, CAS_i chooses $D_0(i)$ if $i < P/2$, otherwise $D_1(P-i)$. As a result, instead of causing rate mismatch between FIFOs, duplicated numbers provide opportunities to re-balance FIFOs. Extremely skewed data actually results in lower stall rates than randomly and uniformly distributed data, as shown in Section V-B.

IV. Parallel Merge-Tree

Using multiple levels of multirate mergers, a parallel merge-tree (PMT) is able to merge multiple sorted sequences simultaneously.

A. *Merging multiple sequences*

An 8-port parallel merge-tree is shown in Fig. 8. When the number of levels $L = \log P$, P input sequences are merged, the input rate for each input sequence is only 1 number/cycle, and the output rate is up to P number/cycle.

Since the area complexity of MM(P) is $P \log P$, the area complexity of a PMT(P):

$$\sum_{i=1}^{\log P} (O(2^i \cdot \log 2^i) \cdot P/2^i) = \sum_{i=1}^{\log P} iP = O(P \log^2 P) \quad (6)$$

Two important observations are revealed here: The accumulated input rate (requirement for memory bandwidth) is equal with the maximum output rate; and the area of parallel merge-trees is scalable to $O(P \log^2 P)$.

B. *Revisiting rate mismatch*

Similar to multirate mergers, a parallel merge-tree has the rate mismatch issue. When numbers are perfectly balanced in all input sequences, the output rate is stable at P number/cycle. However, when numbers are distributed unevenly, the output rate is reduced due to stalls. In the worst case, when numbers are forced to be read sequentially from one sequence to another, the average output rate is dropped to 1 number/cycle. The proposed parallel merge-tree is only suitable for evenly or randomly distributed data sets. A pre-sorted data set needs to be randomly reshuffled before sent to a parallel merge-tree.

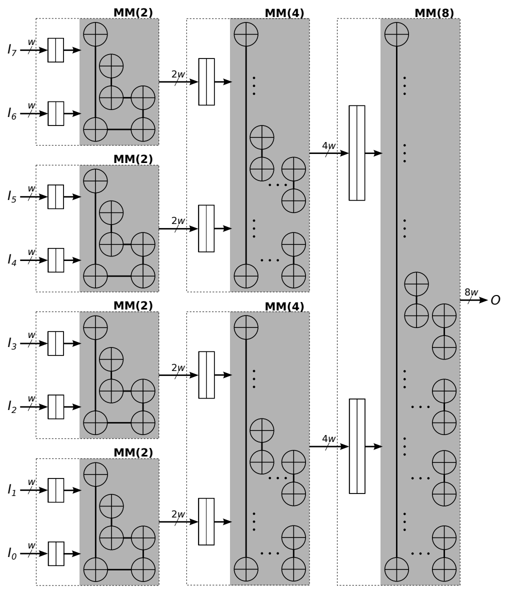


Fig. 8. PMT(8), an 8-port parallel merge-tree.

For randomly distributed data sets, results in Section V-A will demonstrate that the stall rate can be reduced to a negligible level by moderately increasing the depth of FIFOs.

As discussed in Section III-F, a solution to avoid stalls is to increase the input rate of multirate mergers, which is used in the high bandwidth sort merge unit [12]. In this case, all the multirate mergers in Fig. 8 are replaced with MM(8)s. This solution has two outstanding drawbacks.

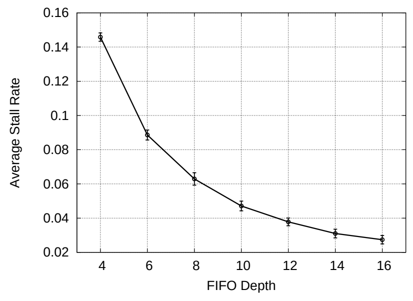
One is the increased area. For a P-port merge-tree, the number of mergers needed is $P - 1$. Therefore, the area complexity increases to:

$$O(P \log P) \cdot (P - 1) = O(P^2 \log P) \quad (7)$$

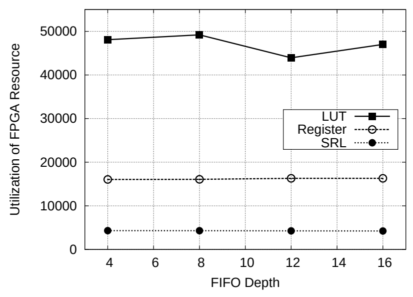
Compared with Equation 6, the area is increased by $P/\log P$ times.

The other drawback, which is even worse, is the significantly increased input data rate. Since the data rate of all input sequences is increased to the maximum output rate, the accumulated input rates of a P-port tree is increased to P^2 number/cycle, P times of the output rate. As reported in the high bandwidth sort merge unit [12], to merge 8 sequences using a 2-level tree, the accumulated memory bandwidth is 6 times of the sorting rate. Clearly this method is not scalable.

We believe allowing mergers to stall is the right choice. The results in this paper demonstrate that the achievable stall rate for most data sets is very small. The benefits of reduced area and memory bandwidth outweigh the slightly reduced sorting rate, and the achievable sorting rate is significantly higher than using the high bandwidth sort merge units.



(a)



(b)

Fig. 9. (a) Stall rate and (b) area with different FIFO depths (a PMT(16) merging 16 sequences of 8K records each).

V. Performance Evaluation

Several parallel merge-trees have been implemented on a Virtex-7 XC7VX485T FPGA [20]. Rather than sorting numbers, practical sorting systems sort records. A record comprises of a key field and an information field. The key field is used in sorting the records while the attached information to each key represents the real data stored in the record. In this paper, all implementations sort records of 64 bits, where both the key field and the information field are 32 bits wide. The stall rate evaluation is based on cycle accurate simulation while the scalability analysis reports post-route results and experiments running on the FPGA. In order to evaluate the pure sorting hardware, we used test pattern generators and a result checker instead of an I/O system.

A. Stall rate evaluation

A multirate merger stalls when one of its FIFOs becomes empty due to imbalanced data distribution. Increasing the depth of FIFOs reduces the probability of stalls. To examine this stall reduction, a PMT(16) is used to merge 16 randomly and uniformly distributed sequences, each of which is 8K records long. The depth of FIFOs is increased from 4 to 16. The average stall rate is defined as the number of stalled cycles divided by the total number of cycles. All test cases are run 10 times for averaging the result. As shown in Fig. 9a, the average stall rate drops significantly at start with shallow FIFOs. Such benefit becomes diminishing with longer FIFOs. Overall, the stall rate is reduced to a negligible level of 0.03 with a moderate depth of 16.

There is no observable area cost of increasing the FIFO depth up to 16 thanks to the implementation described in Section III-D. Fig. 9b reveals the utilization of FPGA resources, including look-up tables (LUTs), registers, and SRLs. Although there is some variation on the total number of LUTs, the utilization of registers and SRLs is constant. To take full advantage of the reduced stall rate, all the remaining implementations use FIFOs of 16 records deep.

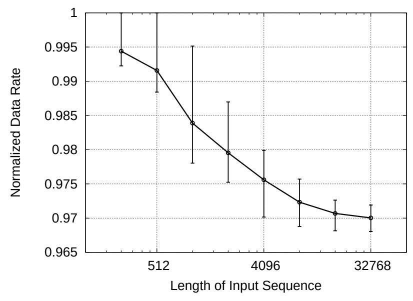


Fig. 10. Normalized data rate of merging sequences with different lengths.

The estimation of stall rate (data rate) is affected by the length of sequences. Fig. 10 shows the normalized data rate (average rate divided by the maximum rate) of merging 16 sequences of different lengths. The estimation error decreases with the increasing sequence length. Interestingly, merging longer sequences results in slightly smaller data rate (higher stall rate). We believe this is due to the higher probability of longer imbalanced segments.

According to the result, sequences of 8K records are chosen for other test cases because they produce relatively accurate data rate estimation and are not too time-consuming to simulate (RTL simulation).

B. Sorting skewed data

Practical data sets are usually skewed. It is important for a sorter to handle skewed data efficiently. An optimization was described in Section III-G, which makes CAS units choosing different inputs to avoid the stalls caused by duplicated numbers. Two PMT(16)s, an unoptimized one (original) and an optimized one, are injected with synthetic data sets generated [29] using 4 skewed distributions: Exponential, Poisson, Pareto and Zipfian. The results are shown in Fig. 11. The optimized parallel merge-tree shows strong tolerance to skewed data while the original parallel merge-tree suffers seriously. Interestingly, when the data is extremely skewed, such as $\lambda \geq 10$ for the exponential distribution and $\alpha \geq 10$ for the Pareto distribution, the optimized parallel merge-tree appears to achieve the maximum rate. We believe FIFOs are easily balanced when most of the numbers are equal.

C. Scalability analysis

It is important for a sorter to be scalable. In the case of parallel merge-trees, the area and the clock period are reasonably scalable with the number of ports (sequences). Table II shows the area and speed performance of various parallel merge-trees. All results are collected post routing.

The utilization of FPGA resource of different parallel merge-trees is depicted in Fig. 12a. The utilization grows

TABLE II
HARDWARE PERFORMANCE OF PARALLEL MERGE-TREES (FIFO DEPTH = 16)

Ports	Period (ns)	Frequency (MHz)	Register	LUT	SRL	Stall Rate	Data Rate (Number/cycle)	Data Rate (Gb/s)
2	4.02	248.5	328 (0.05%)	853 (0.28%)	132	1.75%	1.97	31.3
4	4.69	213.1	1534 (0.25%)	4278 (1.41%)	528	2.16%	3.91	53.4
8	6.12	163.3	5287 (0.87%)	16016 (5.28%)	1608	2.56%	7.80	81.5
16	7.50	133.4	16299 (2.68%)	47001 (15.48%)	4238	2.74%	15.56	132.9
32	10.08	99.2	45445 (7.48%)	142179 (46.83%)	11379	2.98%	31.05	197.1

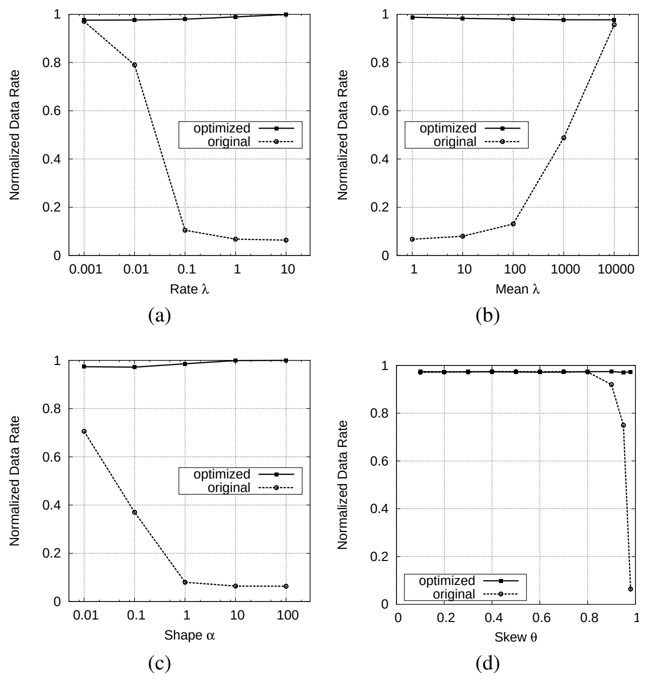


Fig. 11. Normalized data rate of merging data sets with different distributions: (a) Exponential, (b) Poisson, (c) Pareto, and (d) Zipfian.

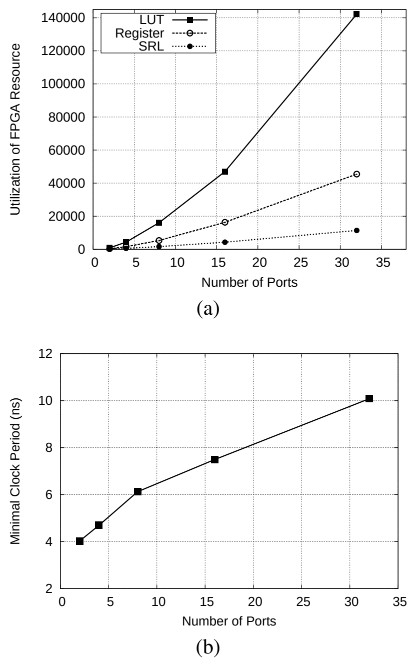


Fig. 12. (a) Area and (b) the minimal clock period of parallel merge-trees with different number of ports (FIFO depth = 16).

slightly more than linear with the number of ports, which complies with the estimation provided in Equation 6.

The maximal clock frequency at which a parallel merge-tree can run is limited by the minimal clock period, which is constrained by the critical path of the root (the largest) multirate merger in the design. Fig. 12b reveals the increasing clock period along with the number of ports. The increasing is slightly slower than linear, which roughly complies with the discussion in Section III-E. In practical implementations, the crossbars in the FIFOs of MM(32) cause routing congestion. To alleviate this issue, the high-radix crossbars are replaced with multi-stage switching networks, which leads to extra levels of logic on the critical path. As a result, the clock period appears linear between 8 to 32 ports.

The achievable data rate (number/cycle) is proportional to the number of ports thanks to the low stall rate. A maximal of 32 sequences can be merged in an FPGA using a PMT(32) at a rate of 31.05 number/cycle or 197.1 Gb/s if considering merging 64-bit records.

D. Reducing the total sorting time

A large data set can be sorted by a merge sorter through multiple passes. Fig. 13 reveals the sorting time of using various parallel merge-trees or a FIFO merge sorter to sort data sets up to 4M records. In all cases, PMT(32) reduces the sorting time by around 160 times compared with the FIFO merge sorter when assuming the same clock frequency. Even if considering merging multiple sequences reduces the clock frequency, the overall execution is always significantly faster than the baseline FIFO merge sorter.

A rough estimation of the sorting time can be calculated as the cycles required in each pass timed by the number of passes. For a FIFO merge sorter, the sorting time is:

$$t_{\text{FIFOMerge}} \sim N \log N \tag{8}$$

as each pass needs N cycles (data rate of 1 number/cycle) and a total of $\log N$ passes are required.

For a PMT(P) which merges P sequences simultaneously

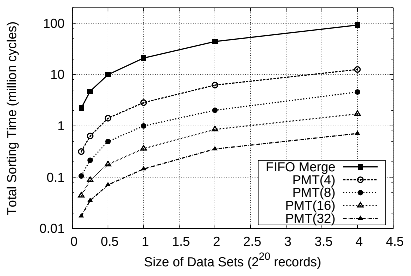


Fig. 13. Sorting time of using different merge sorters.

in each pass, the sorting time can be estimated as:

$$t_{\mathrm{PMT(P)}} \sim (1 - \bar{r}_{\mathrm{stall}}) \cdot \frac{N}{P} \log_P N \tag{9}$$

$$= \frac{(1 - \bar{r}_{\mathrm{stall}})}{P \log P} \cdot N \log N \tag{10}$$

where $\bar{r}_{\mathrm{stall}}$ is the average stall rate, $(1-\bar{r}_{\mathrm{stall}}) \cdot \frac{N}{P}$ is the number of cycles used in each pass, and $\log_P N$ is the number of passes needed. When the stall rate is low, the speed up of using a PMT(P) approaches $P \log P$, which is 160 for a PMT(32).

A significant observation here is the speed up achieved by a PMT(P) is not P but $P \log P$ because merging P sequences simultaneously reduces the number of passes by $\log P$ times. This is an important advantage. The software parallel merge sorter [15] achieves p times of speed up by using p hardware threads. The performance of a PMT(32) is comparable to a software parallel merge sorter running on 160 threads. Considering software sorters need multiple cycles to accomplish each sort operation, hardware sorter may deliver better speed up in the near future.

Another benefit of the reduced number of passes is the save of energy on data transmission. For large scale data sets, the subset sorted in each sorting server is larger than the last-level cache (for multiprocessor/GPGPU) or on-board memory (FPGA). The whole data set is thus read and written from/to the main memory or even solid state disk at least once in each pass. Reducing the number of passes also reduces the number of transmissions between memory and caches, which is a big energy saving.

VI. Conclusion

A scalable parallel merge-tree is proposed in this paper. Different from the state-of-the-art parallel mergers which enforce constant sorting rate, the proposed parallel merge-tree allows stalls and variable sorting rate. This change eliminates the aggressive requirement of memory bandwidth and the unscalable area consumption of the latest parallel mergers. The optimized parallel merge-tree demonstrates strong tolerance to skewed data and provides nearly perfect sorting rate for all random data sets. A 32-port parallel merge-tree, PMT(32), is implemented in a Xilinx Virtex-7 FPGA. It merges 32 sorted sequences simultaneously achieving a data rate of 31.05 number/cycle or 197.1 Gb/s if considering sorting 64-bit records. Compared with traditional FIFO merge sorters, the PMT(32) provides a speed up approaching 160 times.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[2] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys*, vol. 24, no. 1, pp. 63–113, March 1992.

[3] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton, "Modular high-throughput and low-latency sorting units for FPGAs in the large hadron collider," *Proc. of Symposium on Application Specific Processors*, pp. 38–45, June 2011.

[4] J. Gray, C. Nyberg, M. Shah, and N. Govindaraju, "Sort benchmark home page." [Online]. Available: http://sortbenchmark.org/

[5] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[6] K. Ø. Arisland, A. C. Aasbø, and A. Nundal, "VLSI parallel shift sort algorithm and design," *Integration, the VLSI Journal*, vol. 2, no. 4, 1984.

[7] N. Tsuda, T. Satoh, and T. Kawada, "A piepline sorting chip," in *Proc. of International Solid-State Circuits Conference*, 1987, pp. 270–271.

[8] A. Farmahini-Farahani, H. J. Duwe, III, M. J. Schulte, and K. Compton, "Modular design of high-throughput, low-latency sorting units," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1389–1402, July 2013.

[9] 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*, December 2009, pp. 431–434.

[10] 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*, February 2011, pp. 45–54.

[11] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, February 2012.

[12] J. Casper and K. Olukotun, "Hardware acceleration of database operations," *Proc. of International Symposium on Field-Programmable Gate Arrays*, pp. 151–160, 2014.

[13] B. Gedik, R. R. Bordawekar, and P. S. Yu, "CellSort: high performance sorting on the cell processor," in *Proc. of International Conference on Very Large Data Bases*, 2007, pp. 1286–1297.

[14] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1313–1324, August 2008.

[15] S. Odeh, O. Green, Z. Mwassi, R. O. S. Ozdag, and Y. Birk, "Merge path - parallel merging made simple," in *Proc. of International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 1611–1618.

[16] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proc. of International Conference on Management of Data*, 2006, pp. 325–336.

[17] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. of International Symposium on Parallel Distributed Processing*, 2009, pp. 1–10.

[18] K. E. Batcher, "Sorting networks and their applications," in *Proc. of the Spring Joint Computer Conference*, 1968, pp. 307–314.

[19] D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon, "A taxonomy of parallel sorting," *ACM Computing Surveys*, vol. 16, no. 3, pp. 287–318, September 1984.

[20] *7 Series FPGAs Overview (v1.17)*, Xilinx, Inc, May 2015.

[21] T. Graves, "GraySort and MinuteSort at Yahoo on Hadoop 0.23," Sort Benchmark, Tech. Rep., May 2013. [Online]. Available: http://sortbenchmark.org/Yahoo2013Sort.pdf

[22] M. Conley, A. Vahdat, and G. Porter, "TritonSort 2014," Sort Benchmark, Tech. Rep., 2014. [Online]. Available: http://sortbenchmark.org/TritonSort2014.pdf

[23] J. Wang, Y. Wu, H. Cai, Z. Tang, Z. Lv, B. Lu, Y. Tao, C. Li, J. Zhou, and H. Tang, "FuxiSort," Sort Benchmark, Tech. Rep., 2015. [Online]. Available: http://sortbenchmark.org/FuxiSort2015.pdf

[24] S. W. Moore and B. T. Graham, "Tagged up/down sorter — a hardware priority queue," *The Computer Journal*, vol. 38, no. 9, pp. 695–703, September 1995.

[25] G.-S. Liu and H.-H. Chen, "Parallel merge module for combining sorted lists," *IEE Proceedings*, vol. 136, no. 3, pp. 161–165, 1989.

[26] R. Marcelino, H. C. Neto, and J. M. P. Cardoso, "Sorting units for FPGA-based embedded systems," in *Distributed Embedded Systems: Design, Middleware and Resources*, ser. IFIP The International Federation for Information Processing. Springer US, 2008, vol. 271, pp. 11–22.

[27] *7 Series FPGAs Configurable Logic Block — User Guide (v1.7)*, Xilinx, Inc, November 2014.

[28] P. Gigliotti, "Implementing barrel shifters using multipliers," Xilinx, Inc., Tech. Rep., August 2004. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp195.pdf

[29] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *Proc. of International Conference on Management of Data*, pp. 243–252, June 1994.