Automatic Design of Arbitrary-Size Approximate Sorting Networks with Error Guarantee

Vojtech Mrazek
Brno University of Technology
Faculty of Information Technology
Centre of Excellence IT4Innovations
Email: imrazek@fit.vutbr.cz

Zdenek Vasicek
Brno University of Technology
Faculty of Information Technology
Centre of Excellence IT4Innovations
Email: vasicek@fit.vutbr.cz

***Abstract*—Despite the fact that hardware sorters offer great performance, they become expensive as the number of inputs increases. In order to address the problem of high-performance and power-efficient computing, we propose a scalable method for construction of power-efficient sorting networks suitable for hardware implementation. The proposed approach exploits the error resilience which is present in many real-world applications such as digital signal processing, biological computing and large-scale scientific computing. The method is based on recursive construction of large sorting networks using smaller instances of approximate sorting networks. The design process is tunable and enables to achieve desired tradeoffs between the accuracy and power consumption or implementation cost. A search-based design method is used to obtain approximate sorting networks. To measure and analyze the accuracy of approximate networks, three data-independent quality metrics are proposed. Namely, guarantee of error probability, worst-case error and error distribution are discussed. A significant improvement in the implementation cost and power consumption was obtained. For example, 20% reduction in power consumption was achieved by introducing a small error in 256-input sorter. The difference in rank is proved to be not worse than 2 with probability at least 99%. In addition to that, it is guaranteed that the worst-case difference is equal to 6.**

I. Introduction

Sorting is one of the most fundamental operations that is widely used in many applications in computer science including digital signal processing, biological computing and large-scale scientific computing [1].

The hardware sorters are typically employed to improve the performance of applications operating over big data sequences. The sorters can be used either to sort a given sequence [2], [3] or to compute quantiles [4], [5]. These operations represent a typical task performed in database systems, machine learning or business intelligence to distill summary information from huge data sets [4]. In these areas, FPGA-based systems have become popular due to their inherent ability to achieve various trade-offs between throughput and power consumption [1], [3].

On the other hand, the sorting is employed in solving completely different problems. Switching networks, multi-access memories and multiprocessors can be implemented using hardware sorters [6]. In addition to that, sigma-delta digital modulators and various sorter-based arithmetic circuits such as adders, exponential, hyperbolic and logarithmic functions have been proposed recently [7].

The hardware sorters can be classified to two main categories – linear sorters and sorting networks [2]. While linear sorters process one element at a time, sorting networks operate in parallel over the input elements. As a consequence of that, the hardware implementation of linear sorters is usually compact but it fails to scale in performance. On the contrary, sorting networks offer great performance but they become expensive as the number of inputs increases.

Several techniques have been proposed to reduce the area and power consumption of sorting networks. For example, Zuluaga et. al. [2] proposed a domain-specific language and compiler that automatically generates hardware implementations of sorting networks with reduced area optimized for latency or throughput. The area reduction was achieved by reusing the common parts of sorting networks. Chen et. al. [1] introduced a concept of streaming permutation network that was obtained by folding the Clos network. The permutation network was used to construct a high-throughput and a low cost architecture. Compared to [2], significantly better memory as well as energy efficiency was achieved.

Although a lot of effort has been put into the improvement of cost of sorters, it has been demonstrated that many applications from signal processing, computer vision and machine learning exhibit an inherent tolerance to errors in computation [8]. As the power consumption become a critical factor for digital designs, inexact or approximate computing seems to be a viable approach to reduce consumption of many real-world systems and improve the overall efficiency of computers.

Despite the fact that many papers have been published in the field of approximate computing, there is no paper that explicitly addressed the problem of trading the quality of hardware sorters for power efficiency even if there is potential for doing that. The only paper that addressed the problem of approximate sorting was introduced by Leighton and Plaxton in early nineties [9]. The authors theoretically proved existence of an n-input sorting circuit of depth $7.44 \log n$ that sorts all but superpolynomially small fraction of the all possible input permutations. Unfortunately, the hardware implementation remains impractical due to the fact that there is a trade-off between the value of the multiplicative constant and the success probability, and a significant increase in the constant is required for practical instance sizes [9].

A. Our contributions elaborated in this paper

We introduce a scalable method for a construction of arbitrary-size approximate sorting networks. In order to build a large approximate sorting network, smaller instances of approximate or accurate sorting networks are employed. The principle of construction based on recursive bitonic algorithm is inherently tunable to the level of accuracy required for a target application because various approximate as well as accurate sorting networks can be combined together. This gives us the opportunity to obtain approximate sorting networks exhibiting various trade-offs between quality and implementation cost.

In order to design small approximate sorting networks having up to 32 inputs, a systematic search-based design method is proposed. The method works in such a way that it starts with a known architecture of accurate sorting network (generated using bitonic algorithm) that is subsequently optimized (i.e. reduced) to meet the target constraints while introducing a minimal error. The constraints specified by designer can include target implementation cost or target power reduction. The obtained approximate networks can either be applied to construct a larger network or employed autonomously.

Traditionally, a randomly generated set of test vectors is applied to assess the quality of an approximate circuit. This approach, unfortunately, provides no guarantee on the error and make it difficult to predict the behavior of an approximate circuit under different conditions (e.g. when different data-width is used or data with different input distribution are processed). In order to address this problem, we have introduced a method that is able to formally prove and guarantee worst-case error. In addition to that, error distribution can be calculated. Both metrics are based on an extension of zero-one and permutation principle.

II. Sorting networks

The concept of sorting networks was originally studied in 1950s by Armstrong, Nelson and O'Connor and deeply elaborated in 1960s by Knuth [10]. Sorting network is defined as a network consisting of a sequence of elementary operations denoted as *compare-and-swap* (CS) operations that sorts all input sequences. A compare-and-swap operation of two elements, a and b, compares a and b and exchanges (if it is necessary) the elements in order to obtain sorted sequence (a', b'), i.e. $a' = \min(a, b)$, $b' = \max(a, b)$. In hardware, CS is implemented using two multiplexers that are controlled by means of a comparator that determines the maximum of the two (see Figure. 1a).

The sequence of compare-swap operations executed by a sorting network depends only on the number of elements to be sorted, not on the values of the elements. It means that the sequence of comparisons is fixed. This fact represents the main advantage of sorting networks because such a structure can be efficiently implemented using a parallel pipelined hardware architecture. Compared to the linear sorters, the sorting networks do not require to implement a control logic.

A. Representation of sorting networks

The sorting networks are composed solely of wires and comparators. In fact, each comparator implements a two-input sorter. In order to represent sorting networks efficiently, Knuth introduced a notation consisting of vertical segments and horizontal wires [10]. Each vertical segment connecting the two elements being compared represents a single CS operation with arrow determining the larger value (see Figure 1b). A horizontal wire represents an element of input sequence and transmits values from place to place. The unsorted elements (inputs) appear on the left and the sorted sequence is obtained on the right, with the smaller input element appearing on the top output and the larger input element appearing on the bottom output. All comparisons that can be performed in parallel represents a single stage.

B. Construction of sorting networks

Sorting networks can be generated from basic sorting algorithms such as bubble or insertion sort. Both algorithms provide structurally equivalent architectures [3]. Unfortunately, networks generated by these approaches are inefficient like their algorithmic counterparts and consist of many comparator elements.

In order to improve the efficacy, various algorithms have been proposed in literature [10]. The Batcher odd-even merge-sort and bitonic sort represent two simple, yet most efficient algorithms. These algorithms produce sorting networks of the same asymptotic complexity $O(n \log^2 n)$ and the same depth $O(\log^2 n)$ which makes them efficient for parallel implementation. Although the bitonic sorters contain a little bit more comparators, they hardware implementation is the preferred one because all signal paths are of the same length. In addition to that, the same number of comparisons is used in each stage.

Bitonic sorting algorithm is based on repeatedly merging two bitonic sequences to form a larger bitonic sequence. A sequence is bitonic if it can be split to two parts such that the first part is monotically increasing and the second part monotically decreasing, or it can be circularly shifted to become so. The merging operation representing a key step of algorithm is called bitonic merge. The input to this operation is a pair of sequences that are sorted in opposite directions, one in ascending order and the other in descending order, so that together they form a bitonic sequence. Bitonic merge takes this bitonic sequence and from it forms a single sorted

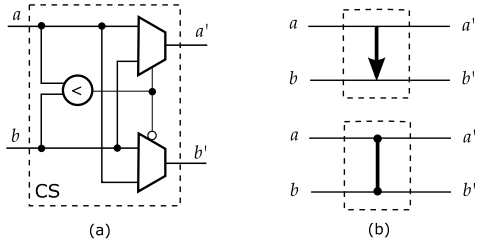

Fig. 1. Compare-and-swap operation: (a) hardware implementation, (b) two equivalent schematic representations using Knuth's notation.

sequence. A complete sorter can be constructed from small bitonic sorters by successively bitonic sorting and merging smaller sequences into larger sequences until we have a bitonic sequence of size n.

In order to reduce high implementation cost, various modifications of bitonic sorting algorithm have been proposed. Stone [11] suggested to employ the perfect shuffle enabling the reuse of $n/2$ processing elements. This approach substantially improved the implementation cost but $\log^2 n$ cycles are required to obtain the sorted sequence. In [12], Lee et al. have improved the time complexity of Stone's algorithm to $\frac{1}{2}\log n(\log n+1)$ introducing an additional logic. The latest modification has been proposed by Chen et.al. [1]. The authors employed a streaming permutation network based on Clos network which is programmable and performs all the data permutations in the bitonic sorting network.

C. Optimal sorting networks

Although a complete and deep theory has been developed around sorting networks, nobody has discovered a sorting algorithm producing the optimal (i.e. minimal) sequence of comparison operators. Even if the best known sorting algorithms such as Bitonic sorting exhibit an optimal asymptotic complexity, there is a large constant factor hidden in the asymptotic bound. The optimal sequence of comparison operators is known only for some instances. The construction of optimal sorting networks is extremely difficult problem even for small number of inputs. For long time, nobody was able to prove the optimality of sorting networks introduced more than 40 years ago by Knuth in [10]. Recently, Bundala and Zavodny proposed a method that is able to construct optimal-depth networks for $n \leq 16$ in reasonable time [13]. Then, Ehlers et al. proved existence of optimal-depth sorting network for $n = 17$ and discovered faster networks for 17, 19 and 20 inputs than the previously known best ones [14]. In [15], Codish et al. proved the optimality of 9-input and 10-input sorting network consisting of 25 and 29 comparators found by Floyd and Waksman in the seventies.

III. Quality of approximate sorters

One of the main issues in the approximate computing is the assessment of quality of approximations. The most popular measure for quality is the mean squared difference between the specification and the output of the approximate circuit that is estimated using a set of test vectors.

The problem of the general quality measures is that they do not assess the quality of sorting process. What's worse, the obtained result depends on a particular set of test vectors. In addition to that, there is no guarantee on the error (e.g. the worst-case error) because only a fraction of all possible input vector was used. In order to address these problems, three data-independent quality measures are introduced in this section.

Let us recall the basic properties of the accurate as well as approximate sorting networks, i.e. comparison networks in general. Let $C(x_1,\ldots,x_n)$ be a comparison network with n inputs, $x_i \in A$ and A be a totally ordered set of elements. It is guaranteed by construction that each comparison network produces a permutation of the input sequence. It means that there exists one to one mapping between the values obtained at the output of comparison network and the values at the input, so no new value can arise during the exchanging performed by compare-and-swap elements. Formally, $C : \pi(x_1,\ldots,x_n) \rightarrow \pi(x_1,\ldots,x_n)$. Hence, every approximate sorting network must produce a partially ordered output for at least one input sequence. In such a case, there must exist at least two outputs that are returning an invalid value.

In general, 2^{wn} input combinations exist to evaluate an n-input sorting network operating with elements encoded using w-bit integers. Clearly, it is intractable to evaluate all possible input combinations, however, the number of input combinations can substantially be reduced by applying the zero-one principle [10] and permutation principle [16].

A. Error probability

According to zero-one principle, 2^n binary sequences are sufficient to determine the error rate. Let $C(\mathbf{x})[i]$ denote value of i-ith output ($1 \leq i \leq n$) of an n-input n-output comparison network C. Let $E_i \subseteq \{0,1\}^n$ be the set of all possible input assignments $\mathbf{x} \in \{0,1\}^n$ for whose an invalid output value is produced, where

$$\begin{aligned} E_i = \quad & \{\mathbf{x} : C(\mathbf{x})[i] = 0 \wedge (x_1 + \ldots + x_n) > (n-i)\} \cup \\ & \{\mathbf{x} : C(\mathbf{x})[i] = 1 \wedge (x_1 + \ldots + x_n) \leq (n-i)\} \end{aligned} \tag{1}$$

Then, $e_i = 2^{-n} \cdot |E_i|$ is the probability that an invalid value is obtained at the i-th output of C. The overall accuracy, i.e. the relative number of correct responses, is equal to

$$accuracy = 1 - \frac{1}{n}\sum_{i=1}^{n} e_i \tag{2}$$

Although it is impractical to determine the size of E_i explicitly by enumerating the assignments satisfying the condition given in Equation 1 (the number of input assignments grows exponentially with the increasing number of inputs n), the number of such assignments, i.e. $|E_i|$, can be calculated easily using a Constaint satisfaction problem (CSP) solver or Binary-Decision Diagrams (BDDs) even for large instances.

Note that the error probability has to be evaluated carefully in practice because there can exist a network with high error rate, but still providing good performance because nearly sorted sequences are produced in most cases.

B. Approximation guarantees

A sorting network can be understood as a structure that computes n quantiles in parallel. The first quantile represents the minimum and the last quantile represents the maximum. Then, we can investigate the difference in rank between the true quantile produced by the accurate sorting network and that of the output produced by the approximate network.

Let us give a simple example. Let $\mathbf{x} = (1, 4, 3, 0, 2)$ be an input sequence, S be sorting network and C be an approximate sorting network producing output $C(\mathbf{x}) = (0, 2, 1, 3, 4)$. It is clear that C returned a partially sorted sequence because the second and third items are invalid, i.e. $C(\mathbf{x})[2] \neq S(\mathbf{x})[2]$ and $C(\mathbf{x})[3] \neq S(\mathbf{x})[3]$. The second output returned the third lowest item of $\mathbf{x}$ (i.e. $C(\mathbf{x})[2] = S(\mathbf{x})[3]$) and the third output returned the second lowest item (i.e. $C(\mathbf{x})[3] = S(\mathbf{x})[2]$). In both cases, the difference in rank is equal to one.

Zero-one principle and CSP solver can be employed to perform formal worst-case error analysis efficiently. Since asymmetric difference in rank may occur at some outputs, it seems to be reasonable to investigate the left (δ^L) and right (δ^R) worst-case distances separately. Firstly, let us define two predicates

$$\begin{aligned} P_L(\mathbf{x}, i, d) &: C(\mathbf{x})[i] = 0 \wedge (x_1 + \cdots + x_n) = n - i + d \\ P_R(\mathbf{x}, i, d) &: C(\mathbf{x})[i] = 1 \wedge (x_1 + \cdots + x_n) = n - i - d + 1 \end{aligned} \tag{3}$$

The problem of the worst-case error analysis can be formulated using Pseudo-Boolean CSP as follows. For each output $i \in \{1, \ldots, n\}$ find maximal $\delta \in \{0, \ldots, n - i - 1\}$ such that $\exists \mathbf{x} \in \{0, 1\}^n : P_L(\mathbf{x}, i, \delta)$. Then, $\delta^L[i] = \delta$ is the left worst-case distance for i-th output. Similarly, the right bound $\delta^R[i]$ can be determined using $P_R(\mathbf{x}, i, \delta)$ instead. Note that it is beneficial to use binary search algorithm to maximize δ because it significantly reduces the number of CSP queries.

Knowledge of the worst-case error alone will not suffice since its probability of occurrence could be negligible. To address this problem, we propose a technique to obtain an error distribution that would provide information about probability of occurrence of errors of different distances. Although it is possible to determine the true error distribution (by counting the number of input assignments that satisfy P_L and P_R, similarly as it was discussed in the previous section), it is practically sufficient and computationally significantly faster to estimate the error distribution. In order to do that, we can adopt permutation principle introduced in [16] and employed to determine the distance between an arbitrary comparator network (i.e. approximate sorting network) and a sorting network. The permutation principle states that it is sufficient to prove response to the permutations of a set consisting of n distinct elements to precisely determine quality of an arbitrary comparison network.

Let us give an example for $n = 3$. Let S be sorting network and C be approximate sorting network consisting of two compare-and-swap operation. Let the first comparator be connected to the first and second horizontal wire, the second comparator be connected to the second and third horizontal wire. In our case, $A = \{1, 2, 3\}$ which gives us six possible permutations that have to be be considered, i.e. $|\pi(A)| = 6$. We can easily determine that $C(\mathbf{x}) = S(\mathbf{x})$ iff $\mathbf{x} \in \pi(A) \setminus \{(2, 3, 1), (3, 2, 1)\}$, i.e. for 4 out of 6 cases. In the remaining two cases, the output of C equals to $(2, 1, 3)$. It means that the first as well as the second output produce erroneous value whose difference in rank is equal to one in both directions (left and right). The results can be summarized using a matrix $\mathcal{H}$ which captures the number of input assignments that cause error at output i whose difference in rank is equal to j. Note that $j = 0$ means that correct response was obtained. The number of correct responses for each output is given in the main diagonal. For example, $h_{3,3} = 1$ because the third output produces always correct result; $h_{1,1} = 4/6$ because there are two cases for that an incorrect response is returned by the first output. The complete $\mathcal{H}(C)$ is as follows:

$$\mathcal{H}(C) = \frac{1}{6} \begin{pmatrix} 4 & 2 & 0 \\ 2 & 4 & 0 \\ 0 & 0 & 6 \end{pmatrix} \tag{4}$$

Interestingly, a relative small subset of all possible permutations is required in practice to obtain a reasonable estimate of error distribution. For example, only 10000 out of more than 10^{77} vectors are required in average to obtain an estimate exhibiting 0.3% relative error (in worst-case) compared to the true error distribution $\mathcal{H}$ computed using BDDs for $n = 256$.

IV. Construction of approximate sorters

In order to construct approximate sorting networks, we propose to modify the Bitonic sorting algorithm as follows.

Algorithm 1: Approximate bitonic sorting

Input: unsorted sequence X, direction $dir \in \{\uparrow, \downarrow\}$
Output: sorted sequence X

```
1  Function sort(dir, X)
2      if |X| = 1 then
3          return X;
4      else if |X| = 2^B then
5          return b-sort(dir, X);
6      else
7          h ← |X| ÷ 2;
8          a ← sort(↑, (x_0, ..., x_{h-1}));
9          b ← sort(↓, (x_h, ..., x_{2h-1}));
10         return merge(dir, (a_0, ..., a_{h-1}, b_0, ..., b_{h-1}));
11 Function merge(dir, X)
12     if |X| = 1 then
13         return X;
14     else if |X| = 2^M then
15         return b-merge(dir, X);
16     else
17         h ← |X| ÷ 2;
18         for i = 0 to h − 1 do
19             if x_i > x_{(h+i)} ⇔ dir =↑ then
20                 swap x_i, x_{(h+i)}
21         a ← merge(dir, (x_0, ..., x_{h-1}));
22         b ← merge(dir, (x_h, ..., x_{2h-1}));
23         return (a_0, ..., a_{h-1}, b_0, ..., b_{h-1});
```

The algorithm consists of two parts – sorting and merging. The input sequence is successively divided into two halves

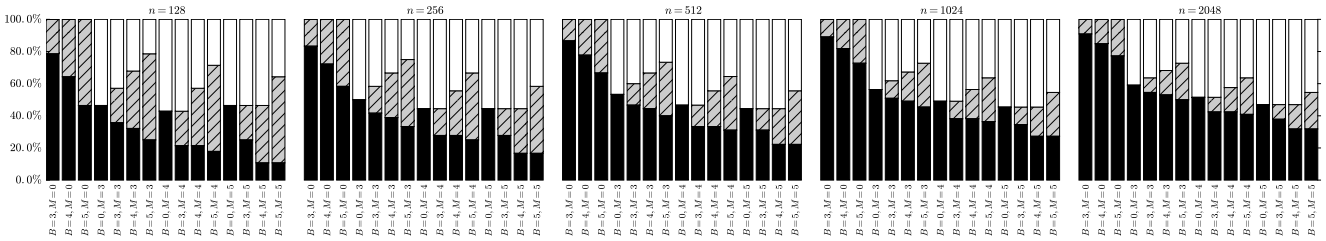

Fig. 2. The relative number of compare-and-swap elements occupied by 2^M-input b-mergers (see ▭) and 2^B-input b-sorters (see ▨) compared to the total number of compare-and-swap elements for various number of inputs n, and selected values of B and M.

until two elements remain. Then, these halves are successively merged until a single sorted sequence is obtained. Two subcircuits can be identified in the resulting sorter – let us call them b-sorters and b-mergers. Our goal is to replace the 2^B-input b-sorters and 2^M-input b-mergers with their approximate versions (see Figure 3). Such a substitution reduces not only the total number of comparisons but may also decrease quality of the sorter. Hence a reasonable trade-off needs to be identified. The designer has the possibility to tune both parameters because various values of B and M can be chosen. In addition to that, approximate networks (i.e. b-sorters and b-mergers) of different quality can be employed.

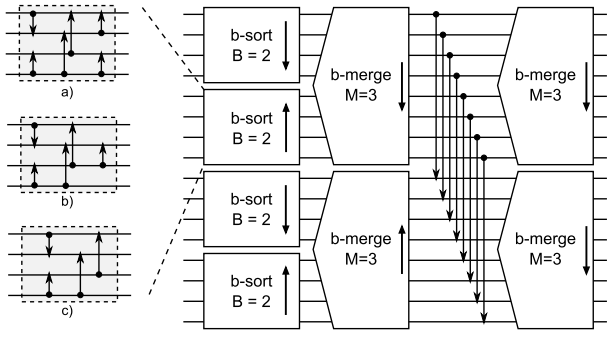

Fig. 3. Sub-circuits (b-sorters and b-mergers) in the 16-input sorter generated using Bitonic algorithm for $B = 2$ and $M = 3$. In addition to that, eight compare-and-swap operations are required. Each b-sorter can be replaced with various 4-input comparison networks, sorter (a) or some approximation (b,c).

As it is non-trivial to predict what values yields the best trade-offs, Figure 2 shows the distribution of compare-and-swap operations for sorting networks having from $n = 128$ to $n = 2048$ inputs. The total number of CSs consists of three groups – the CSs that are required to implement 2^B-input b-sorters, the CSs that are required to implement 2^M-input b-mergers and the remaining ones. The distribution shows the possible area reduction that can be achieved by choosing various values of B and M The upper-bound is limited by the number of comparators that can't be approximated. For $n = 128$, $B = 5$ and $M = 0$, for example, 27% reduction in the total number of CSs can be achieved when the 32-input b-sorters are replaced with their approximate versions occupying half of the resources. Such a reduction is possible because the b-sorters comprise 54% of the total number of CSs.

A. Design of approximate b-sorters and b-mergers

As evident, there is a great potential for improvement in the implementation costs as well as power consumption of sorters especially when larger values of B and M are employed. The only problem is how to obtain high-quality approximations of b-mergers and b-sorters blocks.

The problem of finding an approximate network can be formulated as a constrained optimization problem where the goal is to find a comparison network C exhibiting maximum quality for a target number of compare-swap operations. In order to solve this problem efficiently, we propose to employ Cartesian genetic programming (CGP) [17]. CGP is easy to implement, it can easily handle constraints, it is naturally multi-objective and high-quality approximate circuits have already been obtained in literature [18].

We propose to apply a two-stage procedure. At the beginning, the designer specifies the target reduction that should be achieved. The first stage starts with an exact and accurate solution (i.e. b-sorter or b-merger). The goal is to gradually modify the initial solution and obtain a reduced network of the target cost. In the second stage, which begins as soon as the target reduction was achieved, the search method reflects not only the implementation cost, but also the quality. The second stage aims to improve the quality as much as possible.

CGP employs a simple population-oriented search method. The λ candidate solutions that forms the population are generated from the parental solution (i.e. the best individual discovered so far) using a mutation operator slightly modifying the candidate solutions (up to h randomly chosen genes are modified). Then, the candidate solutions are evaluated and each member receives the so-called fitness score. The highest-scored individual becomes a new parent of the next population. The fitness function $F(C)$ summarize the quality of candidate solutions into a single value and is defined as follows:

$$F(C) = -\begin{cases} \infty & \text{if constraint is violated} \\ \sum_{i,j=1}^{n} h_{i,j}(i-j)^2 & \text{otherwise,} \end{cases} \tag{5}$$

where $h_{i,j}$ is an element of the error matrix $\mathcal{H}(C)$ calculated for a candidate comparison network C. The constraint is represented by the target number of compare-swap operation. The fitness function is designed in such a way that the individuals of higher quality receive higher score. In addition to that,

TABLE I
PARAMETERS OF ACCURATE AND FIVE APPROXIMATE 16-INPUT SORTERS

Impl.	CSs	Depth	Quality indicators						FPGA Synthesis results			ASIC Synthesis results	
	N	D	*accuracy*	Δ_{avg}	Δ_{95}	Δ_{99}	Δ^L	Δ^R	#LUTs	#REGs	Power (W)	Area (μm^2)	Power (mW)
C1	60 (100%)	10	100%	0.00	0	0	0	0	769 (100%)	1120 (100%)	0.24 (100%)	68945 (100%)	1.22 (100%)
C2	49 (81%)	10	49%	0.63	2	3	6	6	621 (81%)	1112 (99%)	0.23 (97%)	58863 (85%)	1.09 (89%)
C3	39 (65%)	10	30%	1.13	3	4	8	8	525 (68%)	1104 (99%)	0.22 (92%)	49698 (72%)	0.96 (79%)
C4	29 (48%)	8	20%	1.70	4	6	9	9	445 (58%)	784 (70%)	0.20 (86%)	37851 (55%)	0.75 (62%)
C5	24 (40%)	5	17%	2.05	5	7	10	10	289 (38%)	640 (57%)	0.20 (82%)	29247 (42%)	0.54 (45%)
C6	19 (31%)	5	16%	2.31	6	8	11	11	253 (33%)	592 (53%)	0.19 (80%)	24664 (36%)	0.48 (40%)

it promotes solutions having narrower error distribution. The fitness score of an accurate b-sorter (b-merger) is equal to zero.

The candidate solutions consists of 2-input elements that can act as compare-and-swap operation or simple buffer. The following encoding of candidate solutions is proposed. Each element is encoded using a triplet consisting of three integers (s, l, f). The first integer $s \in \{1, \ldots, n\}$ defines the index of a horizontal wire where the first input is connected to. The second integer $l \in \{1, \ldots, n-s\}$ determines the length of the corresponding vertical segment. The value, in fact, indirectly encodes the index of horizontal wire where the second input is connected to. Finally, the last integer $f \in \{0^{noop}, 1^{\uparrow}, 2^{\downarrow}\}$ determines the operation of the encoded element. In case that f=0, the element is treated as empty operation and is ignored. Otherwise, a compare-and-swap operation with a given direction is utilized. ASAP scheduling is employed to obtain corresponding comparison network. The proposed encoding guarantee that a valid comparison network is always captured. The number of triplets is defined by the initial solution and remains fixed during the whole search process. The mutation operator modifies values of up to h randomly chosen integers.

The sorting network shown in Figure 3a can be encoded, for example, using 8 triplets as (0,1,1$^{\downarrow}$) (2,1,2$^{\uparrow}$) (1,3,0^{noop}) (1,2,2$^{\uparrow}$) (0,2,2$^{\uparrow}$) (1,3,1$^{\downarrow}$) (2,3,0^{noop}) (2,1,2$^{\uparrow}$). Note that there are two inactive elements (encoded by 3rd and 7th triplet) that do not have any impact on the structure of resulting network.

V. EXPERIMENTAL RESULTS

Firstly, we approximated small sorting networks. In particular, 8, 16 and 32 inputs were considered. The number of inputs was chosen in accordance with the analysis shown in Figure 2 and corresponds with $B = 3, 4, 5$. The search algorithm uses population consisting of $\lambda = 20$ individuals. Up to $h = 5$ integers are modified by mutation operator. The fitness function is calculated using 64×10^4, 256×10^4 and 1024×10^4 randomly generated permutations for $n = 8, 16, 32$, respectively. The optimization process is terminated either when no improvement in fitness score is achieved within the last 15 minutes or the maximum amount of time (2 hours) is exhausted. In case of 8-input (16-input) sorters, the optimization was initialized with the optimal known sorter consisting of 19 (60) compare-and-swap operations. The reference 32-input sorter was obtained using Bitonic sorting algorithm. Ten design points (i.e. design constraint) are considered for each problem instance. The goal is to find approximate sorters consisting of about 95%, 90%, 80%, 70%, 60%, 50%, 40%, 30% and 20% compare-and-swap operations compared to the number of operations of the initial solutions. In addition to that, it is requested that the depth of the approximate sorters is not worse than the depth of the initial accurate sorter.

In total, 30 independent experimental runs were performed and more than $3 \cdot 9 \cdot 30 = 810$ unique approximate solutions were discovered. The Pareto-optimal solutions were identified and implemented as fully streaming pipeline architectures in Virtex-7 FPGA XC7VX330T using Xilinx Vivado and as 45nm VLSI circuits using Cadence Encounter. Then, we conducted the post-place-and-route power analysis performed at 250 MHz (900 MHz for ASIC). Switching activity analysis was employed to improve the accuracy of power estimation.

Due to the limited space, let us discuss the results obtained for 16-input instance. Table I summarizes the parameters of the accurate (C1) and five chosen Pareto-optimal approximate (C2-C6) sorters. Namely, it contains the number of compare-and-swap operations (N), depth (D), quality indicators and sythesis results for FPGA and VLSI. The quality indicators include the accuracy and five differences in rank – mean (Δ_{avg}), 0.95-quantile (Δ_{95}), 0.99-quantile (Δ_{99}) and both worst-cases (Δ^L and Δ^R); all determined over all outputs. The percentages in parentheses indicate the ratio of the value in that column compared to the accurate sorter. As evident, the accuracy gradually decreases with the increasing amount of removed compare-and-swap operations. The same observation is also valid for the implementation cost in ASIC as well as FPGA (see the number of LUT tables and the number of registers). As expected, the implementation cost correlates with N in both cases. A slightly different situation is in the case of power consumption. Because the 16-input sorters are relative small circuits, static part of power consumption dominates in FPGA the dynamic one. As a consequence of that, only 20% improvement in power consumption was achieved for implementation C6 despite more than 50% reduction in implementation cost. No such discrepancy is observable for sorters implemented as ASIC.

Figure 4 depicts the quality matrices of the discovered implementations. Contrasted to the quality indicators, the quality matrices helps to better understand the quality of approximations. The interpretation of $\mathcal{H}$ is as follows. All possible input sequences are correctly sorted in the case of exact sorter (implementation C1). In all cases, the i-th output

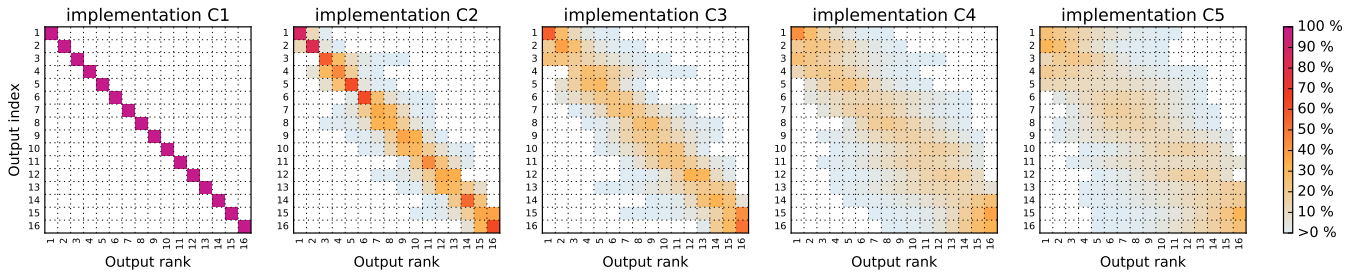

Fig. 4. Quality matrices $\mathcal{H}$ of accurate (C1) and four approximate (C2–C5) 16-input sorting networks

returned the i-th smallest element. It means that only main diagonal contains non-zero values.

Let us discuss the implementation C2 consisting of 49 compare-and-swap operations, i.e. about 18% less compared to the exact sorter. According to the eight column of $\mathcal{H}$, the eight smallest element (i.e. the element with rank 8) is returned by the eight output with probability 33%. This element, however, can be for some input sequence returned also by the output seven (or nine) in 29.3% (17.9%) cases. It means that the difference in one rank occurs in 47.2% in practice. Difference in two ranks (elements is returned by 6th or 10th output) occurs in 19.8% cases. According to the first row of $\mathcal{H}$, it can be determined that the minimum is correctly identified in more than 87.4% cases. In the rest of the cases, the second or third smallest element is returned with prob. 11.8% and 0.8%, respectively. We can conclude that this approximation is of a high quality despite the fact that the worst-case error is equal to 6. According to the quality matrix and Δ_{99}, the difference in rank is not worse than 3 for more than 99% input sequences.

The same principle was applied to approximate 8-input, 16-input and 32-input b-mergers. The initial accurate b-mergers were extracted from the sorters generated using Bitonic sorting algorithm. The experimental setup was kept the same as in the case of sorters. Only the test vectors utilized in the fitness functions are generated in different way. Since two sorted sequences are expected at the input of each merger, it is not necessary to test all the input permutations. Only $\binom{n}{\frac{n}{2}}$ input sequences are required to exhaustively evaluate the quality.

A. Construction of large approximate sorters

The approximate sorters and mergers discovered in the previous experiments were used to construct large approximate sorters for $n = 256, 512, 1024$ and 2048 inputs. The sorters were constructed according to the Algorithm 1. The following parameters were considered: $B = \{0, 3, 4, 5\}$ and $M = \{0, 3, 4, 5\}$. To simplify the problem, ten Pareto-optimal implementations of various sorter (merger) instances were chosen to act as b-sorter (b-merger). In total, 121 architectures were generated and analyzed for each n.

The results for $n = 256$ are summarized in Table II. In addition to the parameters discussed earlier, the value of B and M parameter and the number of compare-and-swap operations required by b-sorters and b-mergers are included. While the whole range of B was utilized, smaller values of M are preferred. It seems that the inaccuracy introduced into the larger mergers has a negative impact on the overall quality. Hence, the majority of architectures employ a variant of 8-input merger (i.e. M=3). The b-sorters represent 10% to 31% the total number of CSs. Compared to the b-sorters, b-mergers occupy significantly larger portion of the CSs especially when the accuracy is higher than 50%. The implementation cost as well as power consumption decreases with decreasing N linearly.

Interestingly, it seems to be nontrivial to predict the quality of the constructed approximate network according to the qual-

TABLE II
PARAMETERS OF ACCURATE AND ELEVEN APPROXIMATE 256-INPUT SORTERS

Impl.	Width		Compare-swap operations			Depth	Quality indicators						FPGA Synthesis results		
	B	M	*sorters*	*mergers*	N	D	*accuracy*	Δ_{avg}	Δ_{95}	Δ_{99}	Δ^L	Δ^R	#LUTs	#REGs	Power (W)
S1	—	—	0%	0%	4608 (100%)	36	100%	0.00	0	0	0	0	55297 (100%)	73728 (100%)	5.997 (100%)
S2_C2	4	4	19%	50%	4112 (89%)	36	92%	0.08	1	1	3	3	49857 (90%)	72320 (98%)	5.433 (91%)
S3	5	4	31%	40%	3880 (84%)	36	85%	0.16	1	1	6	6	47713 (86%)	70784 (96%)	5.154 (86%)
S4	3	3	11%	36%	3584 (78%)	35	60%	0.44	1	2	6	4	44033 (80%)	69376 (94%)	4.826 (80%)
S5	5	3	24%	29%	3288 (71%)	35	47%	0.93	3	6	9	10	41985 (76%)	67328 (91%)	4.505 (75%)
S6	5	3	25%	27%	3192 (69%)	38	32%	1.24	4	6	10	11	41449 (75%)	67272 (91%)	4.424 (74%)
S7_C6	4	3	10%	33%	3120 (68%)	33	28%	1.42	4	6	15	13	38977 (70%)	64256 (87%)	4.238 (71%)
S8	5	3	22%	26%	2936 (64%)	35	20%	1.78	5	7	14	17	38497 (70%)	62400 (85%)	4.012 (67%)
S9_C5	4	3	14%	19%	2688 (58%)	27	18%	2.05	6	8	20	16	32491 (59%)	50924 (69%)	3.601 (60%)
S10_C5	4	3	15%	15%	2560 (56%)	27	12%	2.42	6	9	20	18	30991 (56%)	46610 (63%)	3.553 (59%)
S11	5	3	14%	17%	2232 (48%)	23	9%	3.70	9	13	32	28	27133 (49%)	43358 (59%)	3.024 (50%)
S12	5	3	15%	13%	2136 (46%)	23	7%	4.27	10	13	32	29	26717 (48%)	40509 (55%)	2.897 (48%)

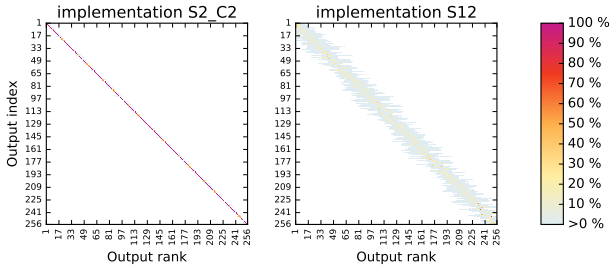

Fig. 5. Quality matrices of two approximate 256-input sorters

ity of the small b-sorters. For example, b-sorters in S2_C2 are implemented using approximate sorters C2. While C2 exhibits only 49% accuracy, S2_C2 provides the correct response for 92% of all possible input sequences. Not only the accuracy was improved, but also the worst-case error is significantly lower (see Δ^L and Δ^R). The same effect is observable also for S7_C6. On the contrary, similar quality is achieved for S9_C5 and S10_C5. These findings suggest that if a b-merger of higher quality is applied, the quality of b-sorter could be improved significantly.

The quality matrices for two chosen implementations are shown in Figure 5. It can be concluded, that the large approximate networks are of high quality even when the number of CSs was reduced significantly. In particular, architecture S2_C2 achieves 10% reduction in power consumption with hardly visible error. In at least 99.9%, the difference in rank is not worse than 1. Implementation S4 exhibiting 20% reduction in power consumption guarantees that the difference is not worse than 2 (5) with probability at least 99% (99.9%), see Δ_{99}. Finally, even implementation S12 could be safely used in many non-critical applications (see Figure 5).

VI. Conclusion and remarks

We addressed the problem of design of approximate sorting networks suitable for hardware implementation exhibiting trade-off between the quality and power consumption. Intuitively, it seems to be sufficient to successively remove the first stages of sorting networks. Unfortunately, our initial experiments revealed that this approach yields non-optimal solutions. Hence, we proposed a scalable method for construction of approximate sorting networks exhibiting trade-off between the quality and power consumption. The method is based on recursive construction of large sorting networks using smaller instances of approximate sorting networks that are designed using a search-based design method.

Many approximate circuits have been proposed in recent years. The correctness, however, is typically guaranteed for precise data and only some estimation is promised for the approximate data [8]. The designers are then reluctant to use such circuits. The strength of our method is that the quality of the approximate networks is guaranteed and formally proved for arbitrary data widths.

Although the power consumption was optimized indirectly by reducing the number of compare-and-swap operations, we have experimentally confirmed that a significant improvement in power consumption can be achieved for sorters implemented not only in FPGAs but also as VLSI circuits. Naturally, the discovered sorters can be employed to improve the computation efficiency of algorithms running on CPUs and GPUs.

Probably due to the lack of a formal apparatus for analysis of approximation guarantees, no survey devoted to the approximate sorters and their applications has been published up to now. Let us mention two applications offering a great space for lowering the computational effort (e.g., energy consumption). Firstly, the small approximate sorters can directly be employed to improve the power consumption of many sorter-based arithmetic circuits or network arbiters [7]. On the other hand, our method can be adopted to produce large networks for power-efficient approximate processing of massive quantile queries, a problem with many real-world applications [5]. In order to do that, the fitness function should reflect the quality of some outputs only.

Acknowledgment

This research was supported by the Czech science foundation project GA16-17538S and Brno University of Technology project FIT-S-14-2297.

References

[1] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on FPGA," in *Proceedings of the 2015 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 240–249.

[2] M. Zuluaga, P. A. Milder, and M. Püschel, "Computer generation of streaming sorting networks," in *Design Automation Conference*, 2012, pp. 1245–1253.

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

[4] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*. New York, USA: ACM, 1998, pp. 426–435.

[5] X. Lin, J. Xu *et al.*, "Approximate processing of massive continuous quantile queries over high-speed data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 5, pp. 683–698, 2006.

[6] K. E. Batcher, "Sorting networks and their applications," in *Proc. of the spring Joint Comp. Conf.*, ser. AFIPS '68. New York, USA: ACM, 1968, pp. 307–314.

[7] H. Fujisaka, T. Kamio, C. J. Ahn *et al.*, "Sorter-based arithmetic circuits for sigma-delta domain signal processing – part I: Addition, approximate transcendental functions, and log-domain operations," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 59, no. 9, pp. 1952–1965, Sept 2012.

[8] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[9] T. Leighton and C. G. Plaxton, "A (fairly) simple circuit that (usually) sorts," in *Proc. 31st Annu. Symp. Foundations of Computer Science*, 1990, pp. 264–274.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[11] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. 20, no. 2, pp. 153–161, Feb. 1971.

[12] J.-D. Lee and K. E. Batcher, "Minimizing communication in the bitonic sort," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 5, pp. 459–474, May 2000.

[13] D. Bundala and J. Závodný, "Optimal sorting networks," in *Proc. of the Language and Automata Theory and Applications: 8th International Conference*. Cham: Springer International Publishing, 2014, pp. 236–247.

[14] T. Ehlers and M. Müller, "New bounds on optimal sorting networks," in *Proc. Evolving Computability: 11th Conference on Computability in Europe, CiE 2015*. Cham: Springer Int. Publishing, 2015, pp. 167–176.

[15] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp, "Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten)," in *26th IEEE Int. Conf. Tools with Artificial Intelligence, ICTAI 2014*, pp. 186–193.

[16] Z. Vasicek and V. Mrazek, "Trading between quality and non-functional properties of median filter in embedded systems," *Genetic Programming and Evolvable Machines*, 2016 *(to be published)*.

[17] J. F. Miller, *Cartesian Genetic Programming*. Springer-Verlag, 2011.

[18] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 432–444, June 2015.