Genet Program Evolvable Mach (2017) 18:45–82
DOI 10.1007/s10710-016-9275-7

Trading between quality and non-functional properties of median filter in embedded systems

Zdenek Vasicek[1] · **Vojtech Mrazek**[1]

Received: 20 December 2015 / Revised: 6 July 2016 / Published online: 19 July 2016

Abstract Genetic improvement has been used to improve functional and non-functional properties of software. In this paper, we propose a new approach that applies a genetic programming (GP)-based genetic improvement to trade between functional and non-functional properties of existing software. The paper investigates possibilities and opportunities for improving non-functional parameters such as execution time, code size, or power consumption of median functions implemented using comparator networks. In general, it is impossible to improve non-functional parameters of the median function without accepting occasional errors in results because optimal implementations are available. In order to address this issue, we proposed a method providing suitable compromises between accuracy, execution time and power consumption. Traditionally, a randomly generated set of test vectors is employed so as to assess the quality of GP individuals. We demonstrated that such an approach may produce biased solutions if the test vectors are generated inappropriately. In order to measure the accuracy of determining a median value and avoid such a bias, we propose and formally analyze new quality metrics which are based on the positional error calculated using the permutation principle introduced in this paper. It is shown that the proposed method enables the discovery of solutions which show a significant improvement in execution time, power consumption, or size with respect to the accurate median function while keeping errors at a moderate level. Non-functional properties of the discovered solutions are estimated using data sets and validated by physical measurements on physical microcontrollers. The benefits of the evolved implementations are demonstrated on two real-

✉ Zdenek Vasicek
vasicek@fit.vutbr.cz

Vojtech Mrazek
imrazek@fit.vutbr.cz

[1] Faculty of Information Technology, IT4Innovations Centre of Excellence, Brno University of Technology, Brno, Czech Republic

world problems—sensor data processing and image processing. It is concluded that data processing software modules offer a great opportunity for genetic improvement. The results revealed that it is not even necessary to determine the median value exactly in many cases which helps to reduce power consumption or increase performance. The discovered implementations of accurate, as well as approximate median functions, are available as C functions for download and can be employed in a custom application (http://www.fit.vutbr.cz/research/groups/ehw/median).

Keywords Genetic programming · Genetic improvement · Cartesian genetic programming · Median function · Comparison network · Permutation principle · Median filter

1 Introduction

Genetic programming (GP) has traditionally been used to evolve entirely new expressions or functions to solve a particular problem which is usually specified by a training data set [23]. With the development of search based software engineering, GP has been applied to repair errors in software and assist in numerous tasks of software engineering [10]. The successful applications of GP in search based software engineering have attracted more and more researchers which has resulted in the establishment of a new research direction called *Genetic improvement* (GI). The Genetic improvement of software is defined as the application of evolutionary and search-based optimization methods with the aim of improving *functional* and/or *non-functional* properties of existing software [38].

The number of lines of code, execution time, memory usage, or power consumption represent the typical *non-functional* properties of software. These properties can be improved, for example, by replacing the existing code fragments by newly evolved code fragments that are semantically equivalent. The improvement of existing software by optimizing its non-functional properties was first addressed by White et al. [38]. Eight different target functions were considered in the paper. The authors showed that GP was able to discover code optimization tricks that are probably unreachable by current compilers. These tricks enabled slight improvements in the execution time of the chosen functions. Recently, Cody-Kenny et al. [3] demonstrated that GP was able to reduce the number of instructions for various manually constructed off-the-shelf implementations of a sort and prefix-code programs written in Java. Genetic Improvement has also been used to evolve an improved version of C++ code using automated code transplantation [22]. The authors evolved a faster version of Boolean satisfiability solver MiniSAT which is specialized for solving a particular problem known as Combinatorial interaction testing. Finally, the reduction of energy consumption of non-trivial programs was addressed in [26]. The authors introduced a system which can further optimize the low level Intel X86 code generated by optimizing compilers. While the previous examples have dealt with non-functional improvements, Langdon and Harman showed that GP can, in addition to non-functional parameters, improve the functionality of existing code [15]. The authors demonstrated that GP was able to automatically improve behaviour (i.e. accuracy) of a widely-used DNA sequencing system consisting of 50,000 lines of C++ code.

A similar research direction has been explored in the field of approximate computing which is a promising approach to obtain energy-efficient computer systems. In constrast to Genetic improvements that preserve the code functionality, approximate computing exploits the fact that many applications are error resilient and do not require a perfect output to be produced. Hence a suitable compromise is sought between the error (quality), power consumption and performance. The approximations can be introduced at the level of hardware as well as software. An approximate solution is typically obtained by a heuristic procedure that modifies the original implementation. For instance, artificial neural networks were used to approximate software modules [7] in order to accelerate computations and reduce power consumption. In addition, search-based methods were allowed to approximate hardware components [21, 36]. Using GP in the context of approximate computing has been reported for digital circuit approximation [29, 34].

In this paper, we deal with GP-based improvements of non-functional properties of programs (C functions) that are intended for low-cost microcontrollers. As we seek significant improvements mainly of power consumption and execution time, we consider the approximate computing scenario and accept some errors in the outputs. The function to be approximated is the median filter which is crucial in signal processing, image processing and sensor data processing. The goal of the GP-based search strategy is to improve the existing, in most cases even functionaly optimal, median implementations and find programs showing suitable compromises between the accuracy, execution time and power consumption for various median functions when implemented on a microcontroller. This paper develops our previous results that were presented at the first GI workshop [20]. In contrast to our previously published work, a more efficient GI method based on a two-stage optimization process is introduced. The quality of the candidate solutions is measured as a distance between the candidate program and a fully-working median implementation. A generalized version of the zero–one principle [14] denoted as *permutation principle* is used to determine this distance. The permutation principle, first introduced in this paper, is formally proven in Sect. 4.2. Compared to the previous results, the quality of the obtained solutions is improved significantly. In addition, the benefit of GI approach is demonstrated using two real-world problems that are typically handled by embedded systems—sensor data processing and image processing. The obtained results are evaluated using data sets and by physical measurements on physical microcontrollers.

In the context of this work, one can observe that the evolutionary design and/or optimization of an (accurate) median outputting program has been carried out by GP only rarely [27]. However, a considerable number of research papers were devoted to the design and optimization of sorting algorithms (e.g., [1, 38]) and sorting networks (e.g. [11, 12, 28, 33]), which are useful structures when the median value has to be obtained. As checking whether a specification (i.e. an original code) and a candidate solution are semantically equivalent is time consuming, exact equivalence checking is not performed in the fitness function. The fitness is usually based on evaluating candidate solutions using a training data set and subsequent testing at the end of evolution using other data sets. According to the zero–one principle, the training data are typically restricted to binary input vectors. A genetic improvement of two different implementations of Bubble-Sort algorithm was demonstrated in [38], where GP enabled the discovery of code optimization tricks probably unreachable by current compilers. These tricks enabled slightly improved execution time of the chosen sorting functions.

The rest of the paper is organized as follows. Section 2 briefly surveys genetic improvement and its relation to the approximate computing. Then an overview of the key areas related to this paper is given in Sect. 3. In particular, the median function and possibilities for the improvement are discussed. In Sect. 4, the permutation principle is introduced. The proposed method is described in Sect. 5. Section 6 introduces the experimental setup. The results are presented and analyzed in Sect. 7. Then, the obtained medians are applied to solve real-world problems. A detailed discussion is given in Sect. 8. Finally, Sect. 9 concludes the paper.

2 From genetic improvement to approximate computing

In contrast to approximate computing that has been developed to improve energy efficiency and performance for the cost of accuracy, GI has always kept the code functionality identical with the original software. In approximate computing, software and hardware is approximated (i.e. simplified with respect to fully accurate implementations) in order to reduce power consumption or increase performance. As a consequence, errors can emerge during computations. In many cases errors can be tolerated because human perception capabilities are limited, no golden solution is available for validation of results, or users are willing to accept some inaccuracies. Therefore, the error (accuracy of computations) can be used as a design metric and traded for area on a chip, delay, throughput, or power consumption.

One way to reduce energy consumption is by allowing timing errors by voltage over scaling or frequency over clocking. Another approximation technique, which is relevant for this paper, is *functional approximation*. The idea of functional approximation is to implement a slightly different function to the original one, provided that the error is acceptable and the non-functional parameters are improved adequately. Functional approximation can be conducted at the level of software as well as hardware.

After introducing several approximate circuits that were created manually [9], researchers started to develop more efficient systematic semi-automatic and fully-automatic methods. EnerJ [24], an extension of Java that adds approximate data types, represents one of the semi-automatic methods. Using these types, the system automatically maps approximate variables to low-power storage, uses low-power operations, and even applies more energy-efficient algorithms provided by the programmer. Axilog is a set of language annotations that provide the necessary syntax and semantics for an approximate hardware design and reuse in Verilog [39]. Axilog enables the designer to relax the accuracy requirements in certain parts of the design, while keeping the critical parts strictly precise. In contrast to fully-automatic methods, an approximate solution is typically obtained by a heuristic procedure that modifies the original implementation. For example, artificial neural networks were proposed in [7] to learn to behave like a general-purpose code written in an imperative language. The trained network then replaced the original code. There are also general search-based methods that allows us to approximate hardware components [21, 36].

While the approximate problem has already been addressed by GP community [30, 34], GI has been used to improve functional and non-functional properties of software so far. However, by having the fitness function of GP-based GI permit errors, one can easily obtain approximate solutions. Applying the GI methodology for approximate computing (particularly for approximate software) seems to be straightforward. The main outcomes would be to obtain better trade-offs among key system parameters (note that the search-based methods are not constrained by various assumptions of mathematically rigorous methodologies) and reducing the optimization time with respect to commonly used solvers such as ILP. The key advantage is that the GI systems can be constructed as multi-objective (i.e. they provide a Pareto front showing the best trade-offs among the error, speed, memory usage, energy consumption, network loading, etc.) at the end of each run.

3 Background

In this section, we give an overview of the key areas related to this paper, especially the median function, construction of median networks and possibilities for the improvement of median functions. The section is concluded with problem formulation.

3.1 Median of a data set

Given a finite sequence of data samples, the median is defined as a value separating the higher half of data samples from the lower half. The median is of central importance in robust statistics [16], as it is the statistic that is the most resistant to outliers that could be presented in a given sequence. Contrasted to the mean, the median is a robust measure of central tendency. The main feature of the robust methods is their high efficiency in a neighbourhood of the assumed statistical model which is widely exploited in signal processing where the median is usually employed to filter the measured data.

There exists two basic approaches to determine the median of a given sequence. A straightforward and naïve approach is to employ a generic sorting algorithm, for example, the most popular and efficient quicksort algorithm. Implementations of sorting algorithms are very compact and robust, however, the execution time needed to determine the median value may vary with the values of the elements in a particular input sequence. This kind of nondeterminism may be problematic in real-time applications intended for microcontrollers having limited computing power. In addition, the sorting of the whole input sequence generates an substantial overhead. In order to eliminate the overhead, a more efficient in-place algorithm known as Quick select can be applied [14].

An alternative way of calculating the median value is to use a *median network*. The median network is a kind of sorting network whose concept is deeply elaborated in [14]. A sorting network is defined as a sequence of elementary compare-swap operations that sorts all input sequences. The sequence of comparators is fixed and depends only on the number of elements to be sorted, not on the values of the elements. Similarly, a median network is a sequence of elementary compare-swap operations that calculates the median for all input sequences. A compare-swap operation of two elements (a, b) compares a and b and exchanges (if it is necessary) the elements in order to obtain a sorted sequence.

3.2 Construction of median networks

A sorting network with n inputs and n outputs can be constructed using an instance of the sorting algorithm which is operating over a sequence of n items. The only condition is that the algorithm must be data independent. Bitonic-sorting and Batcher's odd-even merge sorting are examples of such algorithms.

A median network can be constructed from a sorting network by removing the useless compare-swap operations (i.e. operations that do not contribute to the output value). Aside from this, an optimal sequence of compare-swap operations is known for some median networks [4, 31]. Generally, the direct design of the median and sorting networks is a nontrivial task, especially for larger values of N.

In order to illustrate the difference among the results produced by various algorithms, let us suppose that we need to construct a 9-median network (i.e. a median network for $n = 9$ inputs). When Bitonic-sorting algoritm is used, we obtain a sequence of 23 operations. The Batcher's odd-even merge sorting produces the median network consisting of 22 operations (see Fig. 1a). The optimal 9-median

```
dtype median9_22(dtype *din)
{
   CS(0,1); CS(3,4); CS(5,6);
   CS(7,8); CS(0,2); CS(5,7);
   CS(6,8); CS(0,3); CS(1,2);
   CS(6,7); CS(0,5); CS(1,4);
   CS(2,3); CS(1,2); CS(3,4);
   CS(1,6); CS(2,7); CS(3,8);
   CS(4,5); CS(2,4); CS(3,6);
   CS(3,4)
   return din[4];
}
```

(a)

```
dtype median9_19(dtype *din)
{
   CS(1,2); CS(4,5); CS(7,8);
   CS(0,1); CS(3,4); CS(6,7);
   CS(0,3); CS(1,2); CS(4,5);
   CS(7,8); CS(3,6); CS(4,7);
   CS(5,8); CS(1,4); CS(2,5);
   CS(4,7); CS(4,2); CS(6,4);
   CS(4,2)

   return din[4];
}
```

(b)

Fig. 1 Two instances of a median network for 9 inputs. The compare-swap operation is implemented using macro CS(a, b) which assigns the lower value to din[a] and higher value to din[b]. One of the possible implementations of CS is the following one:

$$\textbf{if } (\mathrm{z} \quad \mathrm{din[b]} - \mathrm{din[a]}) < 0 \textbf{ then } \mathrm{din[a]} \leftarrow \mathrm{din[a]} + \mathrm{z};$$
$$\mathrm{din[b]} \leftarrow \mathrm{din[b]} - \mathrm{z}; \textbf{ end}$$

a Median constructed using Batcher's odd-even merge sorting. **b** Optimal implementation of 9-input median [4] network consists, however, of 19 operations (see Fig. 1b). The corresponding codes in C language are shown in Fig. 1. Note that various implementations can be utilized. If there is a requirement to preserve the input data samples (i.e. to avoid the usage of in-place computations), two temporary variables have to be associated with each output of the CS operation.

Alternativelly, each compare-swap operation can be replaced by two basic operations—minimum and maximum. This allows us to furthermore decrease the total number of required instructions because not every output value calculated by the compare-swap element is subsequently utilized. For example, the last operation shown in Fig. 1b calculates din[2] and din[4]. It is evident that it makes no sense to determine the value of din[2] because only din[4] is returned at the end. The representation based on the minimum/maximum operations enables us to reduce the code size of the 9-median shown in Fig. 1b by 21 % provided that the minimum and maximum macros share the code required to determine the relation between both input values.

3.3 Power-aware improvement of median networks

It is clear that the performance as well as power consumption of a particular median network implementation directly depends on the number of operations a given median network consists of. The higher number of operations results in a higher power consumption as well as a longer execution time. This relation can easily be revealed, for example, by inspecting the implementations shown in Fig. 1. The median networks shown consist of a fixed number of operations. Each operation is executed in the same number of clock cycles on average if it is measured at the level of machine code instructions.

Decreasing the number of operations represents the only way to improve the performance and power consumption. Unfortunately, a reduction of the number of operations is not possible without accepting some errors in the outputs produced by a median function. In other words, we have to search for a sequence of compare-swap operations that are capable of approximating the median. Let us call such a sequence a comparator network.

Two possible approaches can be applied to achieve our goal. One way to obtain an improved median network with a reduced number of operations is to construct a comparator network completely from scratch. It means to employ a variant of GP (e.g. linear GP, cartesian GP, etc.) and evolve programs satisfying the required quality as well as target size constraints (i.e. consisting of the required number of operations). The other possibility is to apply evolutionary techniques to reduce the number of operations of already existing median networks provided that the quality is maximized. In this scenario, a fully working median network used as a starting point is gradually modified according to the genetic improvement methodology. At the end of this process, a comparator network of the highest possible quality consisting of the required number of operations is expected. The question is which of these approaches performs better.

It seems to be natural to employ the first approach, however, due to the limited scalability of the evolutionary design, this approach seems to be extremely inefficient. As shown in [34], randomly seeded GP discovered fully functional solutions for the 9-median, however, no correct solution was discovered for the 25-median. While the evolutionary design of a 9-median is a relatively simple problem, a 25-median consisting of more than 200 operations seems to be outside of the range of possibilities of the evolutionary design approach. If a median consisting of more than 100 operations is required, then direct evolution is unable to accomplish the goal. The authors claimed that solving the larger instances from scratch seems to be impossible for any evolutionary algorithm based on direct encoding.

3.4 Problem formulation

Given an existing median network N, i.e. a sequence of compare-swap operations of length n, and the target number of compare-swap operations m, find an alternative sequence of compare-swap operations M of length m s that this sequence maximizes the functional objective (quality) and minimizes the non-functional objectives.

In general, the problem can be understood as a single-objective as well as a multiple-objective optimization problem. The number of operations and time of execution and power consumption represent the typical software-related non-functional objectives. In addition, the number of stages required to determine the output value can be considered. This parameter is, however, important only when the comparator networks are intended for hardware implementation.

4 The quality of the improved median networks

In this section, we give an overview of approaches that enable us to assess the quality of partially working software and hardware. Then we discuss how to determine the quality of the improved median networks. We introduce and prove the permutation principle which gives a clue on how to determine the quality of median functions efficiently. In order to measure the distance between an original and improved version of the median, a problem-specific quality metric is proposed.

4.1 Common quality metrics

Various approaches to evaluate the quality of partially working software and hardware have been proposed in the literature.

The *error probability* (error rate) and *Hamming distance* represent metrics typically used to measure the quality of digital circuits. The error rate (Hamming distance) is defined as the percentage of input vectors (bits of output) for which the approximate output differs from the original one. In general, 2^{wn} input combinations exist for an n-input median 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. The zero–one principle states that if a sorting network with n inputs sorts all 2^n input sequences of 0's and 1's into a nondecreasing order, it will sort any arbitrary sequence of n elements into a nondecreasing order [14]. As a consequence, 2^n input combinations are sufficient to determine the error rate. Unfortunately, it seems to be difficult to apply this metric in practice. For example, there can exist a candidate implementation slightly modifying one half of the output values, but still providing good performance if used, for example, in image filtering.

The *average error magnitude* is another metric which is used for determining the quality of arithmetic circuits, not only in the field of evolutionary design, but also in the approximate computing. The average error magnitude is defined as the sum of absolute differences in magnitude between the original and approximate circuits, averaged over all inputs. Two complex issues are, however, connected with this parameter when used to evaluate the quality of a median network. Firstly, it is not possible to apply the zero–one principle in this case. As a consequence, we are unable to determine the exact value of this metric. In practice, we have to use a subset of all possible input combinations which helps us to estimate the value of the average error magnitude. The selection of the input vectors must be done carefully because it influences the precision of the estimate. Secondly, the averaging may hide situations in which completely wrong results are returned.

The common problem of the previously discussed generic metrics is that they do not reflect the quality of selecting the median value. In order to investigate the impact of the approximations on the quality of obtained results, regardless of the values of the input items, we introduce a new problem-specific metric. Let us recall that the median of a finite list of numbers can be found by arranging all the numbers from the lowest value to the highest value and picking the middle one. In other words, the median of a finite list of numbers consisting of $2k + 1$ items is equal to the $(k + 1)$th lowest value. The most important property of the median functions implemented in accordance with Sect. 3.2 is that the output always equals one of the input values. Let the output value equal to the jth lowest value. To describe the quality of an approximate median function, we can introduce *distance error* defined as the distance of the item chosen as the output value (i.e. jth lowest value) from the median (i.e. $(k + 1)$th lowest value) calculated as $|j - k + 1|$. Two additional metrics can be inferred from the distance error: *average distance error* defined as the sum of error distances averaged over all input combinations producing an invalid output value and *worst case distance error* defined as the maximal distance error calculated over all input combinations.

Note that it is not necessary to investigate all possible input combinations in practice. The *permutation principle* introduced in Sect. 4.2 permits one to substantially reduce the total number of input combinations that has to be investigated for a given comparator network in order to precisely determine the properties of the network. According to the permutation principle, the aforementioned distance errors can be determined using the permutations of a set S consisting of $2k + 1$ different values. To determine the quality, we propose to use a set $S = \{-k, -k + 1, \ldots, 0, \ldots, k - 1, k\}$. The set S consists of $2k + 1$ successive integers starting at the value $-k$. This particular arrangements enable to calculate the average distance error in the same way as the average error magnitude. This is possible because the median of S is equal to zero and the distance between jth lowest item (i.e. the value $j - (k + 1)$) and $(k + 1)$th lowest item (i.e. median of S)

is equal to $j - (k + 1)$. Compared to the process of determining the average error magnitude, however, a substantially lower number of input combinations is required to be processed by a candidate median implementation.

4.2 The permutation principle

Definition 1 Let Σ be an ordered alphabet. A *comparator network* is a directed acyclic graph with n inputs and n outputs ($n \geq 2$), where each node has two inputs (x_1, x_2) and two outputs (y_1, y_2). The function of a node is defined as $y_1 = min(x_1, x_2) \wedge y_2 = max(x_1, x_2)$, where $x_1, x_2 \in \Sigma$.

Definition 2 A *sorting network* is a comparator network that monotically sorts every input sequence.

Definition 3 Let $A = (a_1, \ldots, a_n)$ be a sequence of n different elements, $A \in \Sigma^*$. Let δ_A: $\Sigma^* \rightarrow \mathbb{N}$ be a mapping which assigns each element $a_i \in A$ the position of this element in the sorted variant of A. Let δ_A be defined as follows:

$$\begin{aligned}
&\delta_A(x) = 0 \Leftrightarrow \quad \forall a \in A{:}x < a \\
&\delta_A(x) = |A| - 1 \Leftrightarrow \quad \forall a \in A{:}x > a \\
&\forall\ 1 \leq i,j \leq n{:}a_i < a_j \Leftrightarrow \delta_A(a) < \delta_A(b)
\end{aligned}$$

For simplicity, let $\delta(\mathrm{A})$ denote the sequence $(\delta_A(a_1), \delta_A(a_2), \ldots, \delta_A(a_n))$.

Lemma 1 ([14]) *Let N be a sorting network with n inputs that transforms a sequence* $A = (a_1, a_2, a_3, \ldots, a_n)$ *to a sequence* $B = (b_1, b_2, b_3, \ldots, b_n)$. *If a monotonic mapping f is applied to the sequence A, the network N transforms a sequence* $A' = (f(a_1), f(a_2), f(a_3), \ldots, f(a_n))$ to $B' = (f(b_1), f(b_2), f(b_3), \ldots, f(b_n))$.

Theorem 1 *Let N be a comparator network with n inputs. Let S be a set consisting of n distinct values. If every permutation of a set S is sorted by N, then every arbitrary sequence is sorted by N.*

Proof Suppose $A = (a_1, \ldots, a_n)$ is an arbitrary sequence which is not sorted by N. This means $N(A) = B = (b_1, \ldots, b_n)$ is unsorted, i.e. there is a position k such that $b_{k+1} < b_k$. Clearly, mapping δ_A is monotonic. By applying Lemma 1 and δ_A, the following holds $\delta_A(b_{k+1}) < \delta_A(b_k)$, i.e. $\delta(B) = \delta(N(A))$ is unsorted. This means that $N(\delta(A))$ is unsorted or, in other words, that the sequence $\delta(A)$ is not sorted by the comparator network N.

We have shown that, if there is an arbitrary sequence A that is not sorted by N, then there is a sequence $\delta(A)$, i.e. a sequence of $(0, \ldots, n-1)$ values, that is not sorted by N. Equivalently, if there is no $(0, \ldots, n-1)$-sequence that is not sorted by N, then there can be no sequence A whatsoever that is not sorted by N. Equivalently again, if all $(0, \ldots, n-1)$-sequences are sorted by N, then all arbitrary sequences are sorted by N.

Clearly, there exists a bijection between all permutations of S and all $(0, \ldots, n-1)$-sequences as follows from the definition of S. In particular, δ_S

ensures the bijective mapping. This means that if all permutations of S are sorted by N, then all arbitrary sequences are sorted by N. □

4.3 The permutation principle and distance error

The permutation principle introduced in the previous section can be employed to determine the distance between an arbitrary comparator network (e.g. partially working sorting network) and a sorting network as follows.

Theorem 2 *Let C be a comparator network, and N be a sorting network, both with n inputs. Let A be an arbitrary sequence $A = (a_1, \ldots, a_n)$. Let $D = C(\delta(A)) - N(\delta(A)) = (d_1, \ldots, d_n)$ be a mapping which assigns each element a_i a number d_i. Then, d_i is error expressed as the number of positions that are required to shift a_i to the right in sequence $C(\delta(A))$ to obtain sorted variant of sequence A.*

Proof Let $B = N(A)$ and $B' = C(A)$. For each element $b'_k \in B'$ holds that difference between a correct position and position of b'_k in a sorted variant of sequence A is equal to $d_k = \delta_A(b'_k) - k$. As N is a sorting network, it holds that $\delta_A(b_k) = k$ which implies that d_k can be expressed as $d_k = \delta_A(b'_k) - k = \delta_A(b'_k) - \delta_A(b_k)$. By applying Lemma 1, it holds that $D = C(\delta(A)) - N(\delta(A))$. □

Definition 4 Let A and B be two sequences of elements. Let $\sim_\delta$ denote an equivalence relation on the set of all sequences defined as follows:

$$A \sim_\delta B \Leftrightarrow |A| = |B| \wedge \delta(A) = \delta(B)$$

To conclude this part, let us give a simple example which illustrates the principle of determining the position error for a comparator network with 4 inputs and 4 outputs and two chosen sequences A and B.

Example 1 Let A and B be two sequences consisting of 4 items defined as $A = (25, 14, 36, 8)$, $B = (16, 12, 20, 2)$. Let N denotes the sorting network and C be a comparator network both with 4 inputs and 4 outputs, where C is defined as follows: $C(a_1, a_2, a_3, a_4) = (min(a_1, a_2), max(a_1, a_2), a_3, a_4)$.

According to the Definition 3, $\delta(A) = (2, 1, 3, 0)$ and $N(\delta_A(A)) = (0, 1, 2, 3)$ which follows from $N(A) = (8, 14, 25, 36)$ where $N(A)$ denotes the sorted sequence A. As the output of C is equal to $C(A) = (14, 25, 36, 8)$, the $C(\delta(A)) = \delta(C(A)) = (1, 2, 3, 0)$. To calculate the positional differences $D_C(A)$, we apply Theorem 2 which yields the following result $D_C(A) = C(\delta(A)) - N(\delta(A)) = (1, 2, 3, 0) - (0, 1, 2, 3) = (1, 1, 1, -3)$. The result can be interpreted in such a way that each of the first three elements of the partially sorted sequence $C(A)$ should be shifted one position to the right and the last element should be shifted three positions to the left. If all the shifts are applied, we obtain a sorted sequence.

The same sequence of steps applied to B yields $D_{C(B)} = C(\delta(B)) - N(\delta(B)) = (1, 2, 3, 0) - (0, 1, 2, 3) = (1, 1, 1, -3)$. It reveals that $D_{C(B)} = D_{C(A)}$, i.e. the same sequence as for A was obtained. It means that we have applied the sequence within the same equivalence class, i.e. $A \sim_{\delta} B$. This fact can be easily checked by comparing the output of $\delta(A)$ and $\delta(B)$. It holds that $\delta(B) = \delta(A)$.

4.4 Final remarks

In general, there exist 2^{wn} input combinations that can be processed by an n-input comparator network operating at w-bits. We have shown that it is sufficient to reduce the number of the possible input combinations to $n!$ to prove the validity of a sorting network due to the existence of permutation principle (see Theorem 1). According to the zero–one principle, the validity of a sorting network can, however, be checked using 2^n binary vectors. As it can easily be checked, the 2^n is for $n \geq 4$ lower than $n!$, hence it seems that the proposed permutation principle does not offer an advantage. However, the problem of zero–one principle is that the binary vectors cannot probably be used to evaluate the quality of a comparator network. The reason is that we are not able to distinguish which value comes from what input (there are only two values—0's and 1's). To address this problem, Theorem 2 helps to determine the so called position error (distance error) which can be used as a basis of an error metric.

The impact of the introduced permutation principle can be seen from theoretical as well as practical point of view. From the theoretical point of view, it was proven that we can use this principle to evaluate the quality of candidate solutions without loss of generality (i.e. it is not necessary to evaluate responses for all w-bit input combinations). The permutation principle significantly reduces the number input vectors that have to be applied to obtain the fitness. In particular, 362, 880 vectors instead of 256^9 vectors are sufficient to precisely determine quality of a 9-input comparator network operating at 8-bits. From the practical point of view, the permutation principle (if properly applied) extremely simplifies the evaluation of candidate solutions because the response of a comparison network (i.e. output value) is equal to the distance from the median value. It means that we can avoid precomputing and storing of a training dataset.

Example 1 illustrates that no additional information about an investigated network is obtained when we try to check some property of a comparator network using sequences belonging to the same equivalence classes. This may happen when randomly generated test cases are used to determine this property. In fact, the randomly generated input sequenced may introduce a bias when used to evaluate quality of a comparison network. The probability of occurrence such cases is relatively high, because only the relation among the values within a generated sequence is important (i.e. not values themselves). Let us give an example. We created 10^6 test vectors consisting of nine randomly generated 8-bit values. Then, we calculated the number of covered equivalence classes according to Definition 4. It revealed that only 337,751 out of 362,880 (i.e. 93 %) of all possible equivalence classes were covered despite the fact that we generated approximately three times more test vectors than the number of equivalence classes. It means that there is many test vectors belonging to the same equivalence class.

The permutation principle and the obtained conclusions can directly be applied not only to comparator networks discussed in this section but also to comparator networks with a single output. In other words, the permutation principle can be used to assess the quality of partially working median as well as sorting networks.

5 The proposed method

In order to search for solutions with some improved level of a non-functional property, we must be able to quantify that property. In our case, the execution time and power consumption are considered. To estimate these non-functional properties, we can use the number of operations of which the median function consists of. This is a fairly reliable high-level estimate not only of execution time but also of power consumption. A more detailed simulation employing an accurate simulator would be necessary in general, however, our programs are designed as a sequence of min and max operations. It is supposed that each operation is transformed by compiler to a sequence of instructions that requires exactly the same number of clock cycles to perform this operation. In addition to that it also reflects the nature of modern embedded systems (e.g. ARM) whose instruction set predominantly consists of instructions that can be executed within one clock cycle.

In this paper, the task is formulated as a single objective optimization problem where the number of operations $\widetilde{n}$ represents a constraint specified by designer. Because both considered non-fuctional objectives linearly depend on this constraint, it is not necessary to include these objectives in the fitness function. This represents the main advantage of the constraint-oriented approach. In addition to this, the constrain-oriented approach is relevant to practice where the designers usually wants to achieve a particular power reduction in order to improve the performance of the whole embedded system.

To achieve our goal, we propose to use cartesian GP (CGP) in its linear form [18]. The linear form seems to be preferred approach compared to the traditional form of CGP representing the solved problems using two-dimensional array of nodes.

5.1 Representation of comparator networks

Each comparator network with n inputs can be represented using a directed acyclic graph consisting of k nodes. In order to encode such a graph, we can map the nodes to a 1D array of N nodes ($N \geq k$) that can be encoded using cartesian GP representation as follows. The number of rows, which is one of CGP parameters, is set to $n_r = 1$. The number of columns n_c is equal to the number of nodes, i.e. $n_c = N$.

The 1D array of nodes can be encoded using a string of integers, the so-called chromosome. The inputs are labelled $(0, 1, \ldots, n-1)$ and the nodes are labelled $(n, n+1, \ldots, n+n_c-1)$. Each node has two inputs and is encoded in the chromosome using three integers—two labels specifying where the node inputs are connected to and a single label specifying the function of the node. Finally, the chromosome contains a single integer specifying the label of a node where the output is connected. The chromosome consists of $3n_c + 1$ integers (i.e. genes).

The main feature of CGP is that n_c as well as n_r (i.e. the total number of nodes N) are constant during evolution. It means that the size of the chromosome is constant because it depends only on n_c. On the other hand, the size of graph represented by this chromosome is variable as some nodes may become inactive. The nodes which do not contribute to the output value are called the inactive nodes. Example is given in Fig. 2 where only 4 out of 5 nodes are active.

Each node can act as minimum or maximum function. The inputs of a node can be connected either to the output of a node placed in previous l columns or to one of the input variables. This restriction ensures that no feedbacks are allowed. The output can be connected to output of any node. The l-back parameter will be unrestricted, i.e. $l = n_c$.

5.2 Quality of candidate solutions

The computational effort of EA directly depends on the number of test cases that are used for fitness evaluation of the GP individuals. Even if we apply the permutation principle which substantially reduces the number of all possible test cases that have to be investigated, we cannot run all of these due to time constraints. Thus, the number of test cases is fixed and specified at the beginning of the evolution. Based on the preliminary experiments and in accordance with observations related to the minimum number of test vectors required to evaluate candidate solutions [34], we determine the number of test cases as $T = 10^3 \times n^2$, where n is the number of input variables. The number of test cases is chosen in such a way that we are able to relatively precisely estimate the quality of the individuals while the time of fitness evaluation remains reasonable. Surprisingly, this simplification does not have any significant effect in practice provided that a reasonable number of unique permutations is used. The example given in Fig. 3 demonstrates how the number of test cases influences the shape of distribution of distance error for an approximate version of 25-input median. If more than 10^4 test vectors are used, the distributions are almost identical.

The first generated test case is the sequence $S = (-k, -k+1, \ldots, 0, \ldots, k-1, k)$ where $n = 2k + 1$. The next test case is obtained by swapping two randomly chosen items. This step ensures that a permutation of S is obtained. Note that the process of

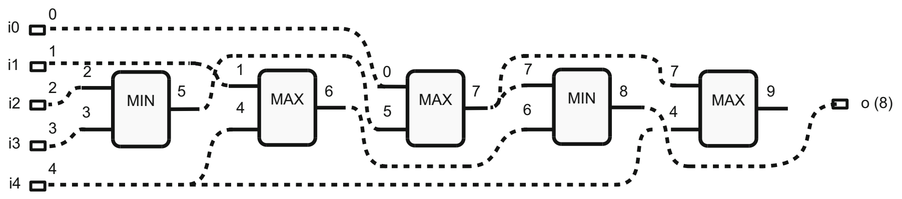

Fig. 2 Example of a 5-input comparator network encoded using cartesian GP with parameters: $k = 5$, $n_c = 5$, $l = 4$. Chromosome: 2, 3, min; 1, 4, max; 0, 5, max; 7, 6, min; 7, 4, max; 8. Node 9 is not used. The behaviour of the encoded comparator network is defined as: $o = \min(\max(i_0, \min(i_2, i_3)), \max(i_1, i_4))$

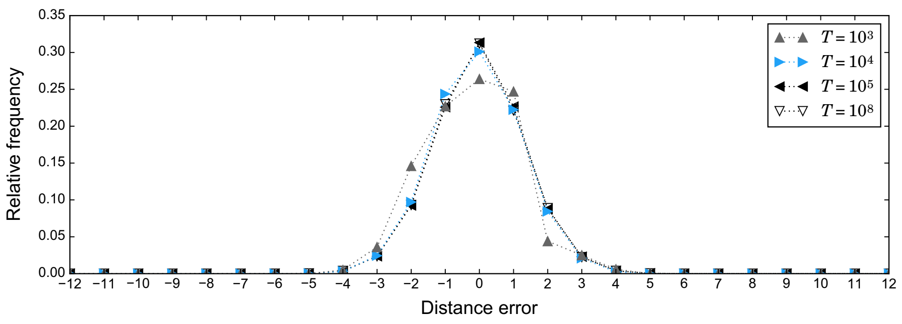

Fig. 3 Distribution of distance error for an approximate version of 25-input median as a function of the number of test cases T

generating the test cases have to be deterministic because we have to guarantee that exactly the same fitness score is obtained for individuals that represent the same behaviour. In order to satisfy this requirement, a separate random generator is used to perform the random exchanges. This generator is reinitialized with the same seed whenever we begin to generate the permutations.

The quality of the GP individuals is determined as follows. For each test case, the chromosome is interpreted. This step requires to successively determine the value at the output of each node. Finally, the response of a comparator network encoded by the individual is calculated. Because the permutations of a set proposed in Sect. 4.1 are applied to the inputs, the obtained response equals to the distance error determined for a given test case. In order to prefer the implementations with the lowest worst case distance error, we propose to calculate a histogram of distance errors and summarize the obtained results as follows:

$$q(C) = h(C,0) - \sum_{i=-k}^{k} h(C,i)i^2, \tag{1}$$

where $q(C)$ denotes the quality of a comparator network C and $h(C, i)$ represents the number of occurrences of a case for which the distance error equals to i, formally:

$$h(C,i) = \sum_{t \in T} \begin{cases} 1, & \mathrm{if C(t) = i.} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

where $C(t)$ denotes the response of the comparator network C obtained for a test case $t \in T$ and T is a set of considered permutations of S. There are two reasons for including i^2. Firstly, only positive numbers are summed. Secondly, a natural weight is provided in order to emphasize the most important part of the histogram.

5.3 Search method

The search method follows the standard CGP approach [18], i.e. the evolutionary strategy $1 + \lambda$ is applied. The initial population is seeded by an existing median network. In order to generate a new population, λ offspring individuals are created by a point mutation operator modifying h genes of the parent individual. The best individual of the current population (i.e. the parent individual together with λ offspring) serves as the parent of new population. The process is repeated until a given number of generations is not exhausted.

One mutation can alter either the function of a node, node input connection, or output connection. If a mutation hits a non-active node, this is detected and the candidate solution is not evaluated in terms of functionality because it has the same fitness as its parent. Mutations that do not affect the fitness score are called neutral and seem to be important in CGP because a series of neutral mutations can accumulate useful structures in the part of the chromosome which is not currently active (see detailed analysis in [8, 19]). In order to support this kind of neutrality, neutral mutants always replace their parent in CGP. One adaptive mutation can then connect these structures with active nodes which could lead to discovering new useful implementations.

In order to obtain an approximate version of median function M, we propose to apply a two-stage procedure. At the beginning, the designer specifies the target reduction that should be achieved, e.g. 15 %. The specified value is internally understood as the number of operations L of the approximate median function. In our example, L is equal to 85 % of the number of operations in the original median function.

The first stage starts with a fully functional solution. As has been discussed in Sect. 3.2, the initial solution can always be obtained in practice. In this stage, the goal is to gradually modify the initial sequence consisting of minimum and maximum operations and produce a reduced sequence of length L providing that a 5 % difference is tolerated with respect to L (tolerating a small deviance in the number of operations is acceptable; otherwise the search could easily stuck in a local extreme). The fitness function fit_1 used in the first stage is thus solely based on the number of operations

$$fit_1(C) = |C|, \tag{3}$$

where C denotes a candidate solution implemented using $|C|$ operations.

In the second stage, which begins after obtaining an implementation consisting of the target number of operations, the fitness function reflects not only the size, but also the quality:

$$fit_2(C) = \begin{cases} q(C), & \text{if } 0.95\text{L} \leq |\text{C}| \leq 1.05\text{L}. \\ -\infty, & \text{otherwise.} \end{cases} \tag{4}$$

It is requested that the number of operations remains within 5 % tolerance with respect to L. Candidate circuits violating this hard constraint are discarded.

The proposed two-stage method eliminates the problem with seeding of initial population which may be considered as a limitation of the resource-oriented method [20]. The advantage of our method is that we do not need to implement a heuristics for generating the initial solution consisting of L operations. Instead, a fully working median function obtained by pruning a sorting-network is used as the start point. In addition to this, it was demonstrated that the randomly seeded CGP was unable to produce reasonable solutions when the complexity of the problem to be solved increases. The role of seeding was investigated for example in [34]. The benefits of the two-stage method are not only in improving the quality of evolved circuits, but also in reducing the time of evolution.

6 Experimental setup

In order to evaluate the performance of the proposed approach, i.e. the ability to improve the considered non-functional parameters of the existing median functions, namely time of execution and power consumption, we have chosen four instances of the median filter that are common in practice. The results of optimization for 9-median, 11-median, 13-median, and 25-median will be reported. While the 9-input and 25-input medians are typically employed in image processing, the 9-input, 11-input and 13-input medians represent instances used to filter data coming from sensors. We did not consider the lower number of inputs because there is nearly no potential for improvement due to the small code complexity.

As previously mentioned, the designer has to specify the target reduction that ought to be achieved by reducing the number of instructions. Eight to eleven design points (i.e. different values of L) were considered for each problem. We carried out 100 repetitions of CGP at each design point to evaluate the variation in the output caused by the random seed. In total, 4000 experimental runs were performed. To be able to evaluate all runs in a reasonable time, the number of generations was limited to $g_{max} = 1 \times 10^4$. The number of generations is based on the initial experiments and represents a compromise between the ability to demonstrate the advantage of the genetic improvement in the solving of the chosen problem and the amount of required computational resources. If the objective is to find the best possible implementation for a certain design point, we recommend to increase the number of generations. In order to improve the efficiency of the fitness function, approach proposed in [35] was employed.

The following settings was used for the search strategy: Twenty offspring individuals are generated from the parent (i.e. $\lambda = 20$) using the mutation operator that modifies up to 5 % of the chromosome genes. The number of columns n_c is fixed for each design point and is initialized according to the number of operations of the original median function. In the case of 9-input and 25-input median, the optimal implementations consisting of the minimal number of compare-swap elements were used from [4]. Each compare-swap element was replaced by minimum and maximum operation and the worthless operations were removed as mentioned in Sect. 3.2. The obtained sequence of minimum and maximum operations was used as a starting point for seeding the evolutionary algorithm. In remaining cases, the initial fully working median networks were derived from a 25-input median network by reducing the number of inputs and removing redundant operations. We have verified that this approach produces more compact median networks compared to the results obtained using the approach employing a sorting algorithm. The parameters of the initial networks are summarized in Table 1.

Table 1 Parameters of the fully working median functions used to seed the evolution and the range in which the design points are sampled

Parameter	9-Median	11-Median	13-Median	25-Median
Number of compare-swaps elements	19	33	43	99
Number of min/max operations	30	56	74	174
Number of min operations	15 (50 %)	28 (50 %)	37 (50 %)	87 (50 %)
Number of max operations	15 (50 %)	28 (50 %)	37 (50 %)	87 (50 %)
Minimum value of L	8	8	10	50
Maximum value of L	30	56	74	174
Number of design points	11	8	8	13

7 Results

The results of the evolution are summarized in Fig. 4. For each problem and each design point, the normalized fitness score is given. This score is calculated according to Eq. 4, however, the results are normalized by the total number of test

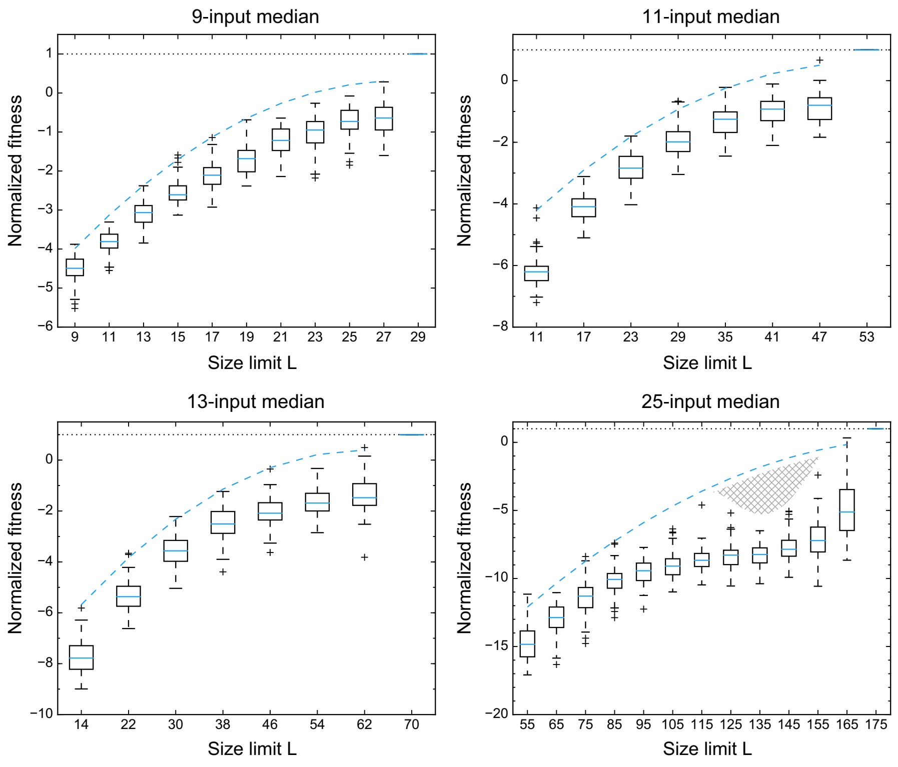

Fig. 4 The fitness score of the evolved comparator networks (approximate median function) based on 100 experimental runs performed for each design point. The *dash line* represents target Pareto frontier

cases. The interpretation of the y-axis is as follows. While the fully working median functions represented by the fittest solutions have their fitness score equal to one, the solutions of lower quality have assigned the fitness score lower than one.

For each problem, we sampled the design space equidistantly to be able to construct the Pareto frontier which helps us to discuss the performance of the method. As mentioned earlier, the maximum value of L is bounded by the size of the initial solution. Conversely, it makes no sense to explore situations where L is lower than the number of input variables because it means that some inputs will not be involved in computation. In the case of 25-input median, we restricted the lower bound even more because it would be computationally expensive to perform evolution for all cases. According to the measurements, 8.8 ms are required in average to calculate fit_2 for 9-input median. This time, however, increases up to 368.3 ms in the case of 25-input median. The experiments were conducted on a 64-bit Linux machine running on Intel Xeon X5670 CPU (2.93 GHz, 12 MB cache) equipped with 32 GB RAM.

Interestingly, compared to the resource-oriented method [34], our method is extremely efficient if the time required to obtain an implementation consisting of L operations is considered. According to the experiment, the average duration of the first stage is less than 10 ms in the case of 9-median and less than 373 ms in the case of 25-median.

The obtained results given in Fig. 4 are presented using boxplots which illustrate distribution of the normalized fitness calculated independently for each considered design point. As it can be seen, the variance of the fitness score is quite low for each design point. Taking into account that the number of generations was relatively low, these results demonstrate the robustness and stability of our method. The only exception is the 25-input median where we can see higher variance primarily at both extremes of L. In order to analyse this situation more thoroughly, we created a target pareto frontier (see dashed lines in Fig. 4) representing the goal of evolution. This Pareto frontier was obtained by interpolation of the fittest implementations obtained for 9-input, 11-input and 13-input median. The obtained regression models were generalized and projected backward to the plots. In most cases, we were able to find solutions that are very close to this imaginary pareto frontier. Unfortunately, in the case of 25-median (see Fig. 4, bottom right), we can see that there are cases in which the fitness score of the obtained results is far from the expected one. This is evident especially for cases where L is between 125 and 155. To investigate the reason of this gap, we tried to prolong the time of evolution for few of these cases and we discovered that this problem is caused by the insufficient number of generations. In order to obtain better results, it would be necessary to increase n_g adequately (at least by two orders of magnitude).

Whilst the initial implementations of 9-input and 25-input median networks remained unchanged, which was in fact expected as it is believed that the corresponding sequences of compare-swap elements are optimal, the evolution discovered improved versions of 11-input fully functional median function consisting of 50 operations and improved version of 13-input fully functional median counting 66 operations which yields 11 % reduction in both cases.

A more detailed analysis of the quality of discovered solutions is shown in Fig. 5 where we present histograms of the error distribution for each problem. The histograms are created using the best solutions obtained from all experimental runs. It means that for each design point, the fittest solution was identified and chosen. The quality is expressed in terms of the distance error. The histogram of distance errors are calculated for each discovered solution using 1000 times more permutations compared to the number of permutations utilized to determine fit_2; this enables to obtain precise results exhibiting the error in the order of 10^{-3}.

Let us discuss, for example, the results for 11-input median (see Fig. 5, top right). If we reduce the number of operations by 12 % (i.e. to 44 operations), the output value is determined correctly in more than 93 % all possible cases. In the rest of the cases (i.e. 6 %), the output value is determined incorrectly as the 4th lowest item of a sorted list of numbers. In less than 0.9 % of cases, the 6th lowest item is returned. Because the median value corresponds with 5th lowest item, the distance between median and output value is equal to 1 in both cases. If the number of operations is reduced to 20 (60 %), the worst case error increases to 2. According to the distribution of errors, this error, which is caused by outputting 3th or 7th lowest item, occurs in 3.6 % of all input cases only. The remaining 47.2 % erroneous outputs are caused by selecting 4th or 6th lowest item.

An interesting feature of the discovered solutions is the asymmetric distribution of the errors. This is more evident if we look at the histogram for 25-input

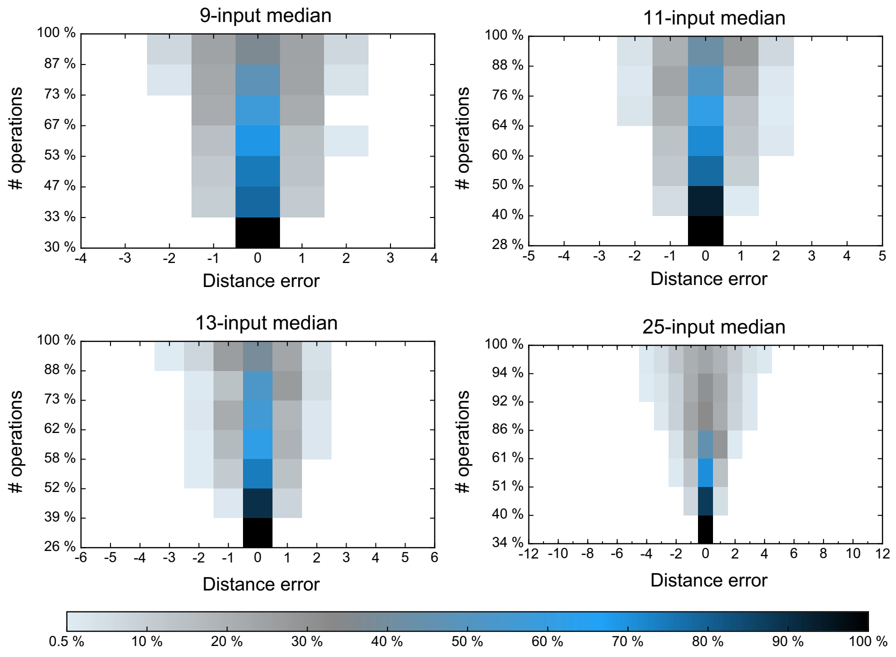

Fig. 5 The quality of the best discovered solutions consisting of a different number of operations expressed in terms of the distance error. The zero error means that the 5th, 6th, 7th, and 13th lowest item of input sequence was returned for 9-input, 11-input, 13-input, and 25-input median respectively

Table 2 Parameters of the improved implementations of 9-input median

No. of operations	Achieved improvement (%)	Error probability(%)	Distance error		
			Mean	Left/right	Worst-case
9	70.00	68.11	0.871 ± 0.702	−2	2
10	66.67	63.36	0.776 ± 0.677	−2	2
14	53.33	53.12	0.591 ± 0.601	−2	2
16	46.67	42.85	0.428 ± 0.495	−1	1
20	33.33	30.93	0.321 ± 0.491	−1	2
22	26.67	25.26	0.253 ± 0.434	−1	1
26	13.33	21.53	0.215 ± 0.411	−1	1

comparator network consisting of 150, i.e. 86 %, operations (see Fig. 5, bottom right). While the 4th lowest item is returned in 20 % of cases, the 6th lowest item is returned in more than 29 % of cases. We have not investigated the exact reason because it does not represent a real problem, however, it is worth noting that we have obtained many solutions with the same fitness score and it may happen that there is a solution with the same or slightly lower fitness score having a symmetric distribution of errors.

We can conclude that the obtained reduced median networks are of high quality. Even in the extreme case, where approximately 50 % of operations are removed, the error is not worse than one position for 9-input median, 2 positions for 11-input median, and 3 positions for 13-input and 25-input median respectively. The 25-input median consisting of more than 170 operations offers the largest possibilities for improvement. We can remove more than 25 % operations without a significant decrease in the quality. For more than 75 % of all possible input combinations, the median value or the values next to the median are returned.

To have a notion of properties of the discussed error metrics, Table 2 reports the error probability, mean distance error, and left and right worst case distance error for 9-input median. It can be seen that as the number of operations decreases, the error probability as well as the distance error are increasing. The mean value increases, however, it is not easy to a priori specify the required target value. The same is valid even for the error probability which gives the number of invalid output values, i.e. the amount of cases in which a value different from median was returned. Looking at the results shown in Fig. 5, it can easily be revealed that the issue with the mean value is that the distribution of errors is not the Gaussian distribution, especially for cases with a small reduction of the number of operations.

8 Improved medians in real embedded systems

Because the medians are typically employed to solve some real problem, we take the best discovered approximated median filters whose quality was discussed in the previous section and evaluated their performance in two different real-world problems—processing of data acquired by sensor devices, and removing of noise in image data.

For each case study, the problem is briefly introduced first. Then, non-functional parameters of evolved as well as commonly used implementations are analysed and discussed. Finally, the impact of the approximate medians on quality and performance is evaluated providing that the approximate medians are employed as the main component which process data.

Four microcontrollers were chosen to evaluate the non-functional parameters of the evolved median functions. The microcontrollers were programmed using the complied C codes of discovered implementations discussed in the previous sections. Two non-functional parameters were measured: (a) the time that each microcontroller spends in a routine which computes the median value, and (b) energy consumed by the microcontroller to execute this routine.

A specific program was implemented, compiled and executed by the microcontrollers to perform the measurements. The program is designed as follows. Firstly, an input vector consisting of n integers is randomly initialized and fed to the routines calculating the median. Note that n is equal to the number of inputs of median. Then, an infinite loop is executed, which contains calling of the routine calculating the median value followed by a code modifying a randomly chosen value of the input vector to another value. Passing one iteration of the loop is indicated by inverting the logic value on a given pin. The execution time is then obtained using an oscilloscope by monitoring the period of the signal on the pin. The average execution time is reported.

In order to precisely determine an average energy needed to calculate the median value, all unused peripheral devices are switched off. Only those external components remain used which are necessary for program execution. Energy consumption was measured using Agilent N6705B DC Power Analyzer displaying the error lower than 0.025 % for voltage as well as current measurements.

8.1 Microcontrollers used for testing

In order to evaluate the non-functional parameters, we have chosen the following common-off-the-shelf microcontrollers available in our lab: 8-bit microcontroller of Microchip PIC family with code name PIC16F628A, 16-bit PIC24F08KA102, low-power 16-bit microcontroller MSP430F2617 from Texas Instruments and 32-bit ARM-based microcontroller STM32F100RB produced by STMicroelectronics. The goal is to present results for various architectures because there typically exist variations in the performance caused by different instruction sets on the one side and different internal architecture on the other side. To be able to interpret the obtained results, the main features of the microcontrollers are briefly discussed in this section.

The 8-bit PIC equipped with 3.5 kB of FLASH and 224 B of RAM is optimized for low-cost applications. Hence, a simple accumulator architecture without a stack is used. The instruction set consists of 35 instructions encoded using a 14-bit wide instruction word. The two-stage instruction pipeline allows all instructions to be executed in a single cycle, except for program branches. The chosen chip has an internal oscillator running at 4 MHz and consuming about 10 nA in the sleep mode and about 565 μA in the active mode. Note that these values were measured when all the peripherals were deactivated.

The 16-bit PIC represents a class of microcontrollers with a register architecture consisting of 16 general-purpose 16-bit registers and 7 special registers. The instructions are encoded using a 24-bit instruction word with a variable length of the opcode field. The chosen chip contains 8 kB of FLASH memory, 1.5 kB of RAM memory and employs an internal oscillator running at 8 MHz. The instructions require from 1 to 3 clock cycles and are executed at 4 MHz. Our chip consumes about 4 mA in the active mode and 25 nA in the sleep mode.

The MSP430F2 is a 16-bit ultralow-power RISC microcontroller with register architecture optimized for processing data from sensor devices. The chosen CPU consists of 16 registers, is equipped with 92 kB of FLASH memory and 4kB of RAM, and can operate at 16 MHz. The calibrated digitally controlled internal oscillator can be configured to generate up to 8 MHz signal for system clock. The instruction set consists of 51 instructions with three formats and seven address modes. In contrast with PIC, there are instructions that enable to access two memory operands. The instructions require from 1 to 6 cycles to be executed. The instructions working with registers require a single clock cycle, the instructions addressing memory require 3 or 6 (when two memory accesses are required) cycles. The chip consumes 365 μA in the active mode at 1 MHz and 500 nA in the standby mode. In order to exploit the low-power capabilities, we configure the internal oscillator to operate at 1 MHz.

The STM32F100RB incorporates a high-performance RISC ARM Cortex M3 core offering twelve 32-bit general-purpose registers. This core builds on the ARMv7-M architecture and shows higher computational power compared to the aforementioned chips. For example, a single-cycle multiplication and a hardware division are supported. STM32 is equipped with 128 kB of FLASH memory, 8 kB of RAM and operates at 24 MHz. The maximum current consumption in the sleep mode is approx. 3.8 mA. When the peripherals are enabled, the current increases to 9.6 mA. The current in active mode ranges from 10 to 150 mA depending on the state of peripherals.

8.2 Evolved code on different microcontrollers

The process of obtaining C code from a chromosome is straightforward. Every active node, starting from one with the lowest index, corresponds with a single line of code containing a call of *min* or *max* function whose operands are taken from the input sequence or the outputs of preceding operations.

The *min* and *max* functions are defined as two macros outputting the minimal and maximal value for two operands. The compiler is then able to unroll the code and optimize it in terms of register assignment and overall performance.

8.3 Processing data from sensor devices with approximated median filters

When we look at signals coming from various devices such as A/D converters, temperature sensors, or accelerometers, the data are noisy even in a perfect environment. In a real situation, where the accelerometers are, for example, used to stabilize various flight vehicles, the situation is even worse because of various vibrations caused by motors or propellers that are for example out of balance. When such a sensor acts as a central element controlling a process, it is necessary to remove the noise so as to prevent unwanted behaviour.

There are many filters that can be applied to smooth the measured data, for example, a variant of *low-pass filter*. The filter tries to keep the low frequency data while removing the high frequency noise (i.e. spikes). A low-pass filter usually is implemented in a situation where a limited number of computational resources are available because its implementation is simple. It can be implemented as an exponentially weighted moving average $x_{t+1} = \alpha y_t + (1 - \alpha)x_t$ where y_t represents data measured at time t and x_t the output value obtained at time t. Alternatively, a more robust Kalman filter may be used [13]. In contrast to the low-pass filter which has a fixed parameter α, Kalman filter is an adaptive estimator which minimizes the mean square error of the estimated parameters according to the previous state and actual measured value. Given only the mean and standard deviation of noise, the Kalman filter is the best linear estimator.

Unfortunately, there are two issues connected with the usage of linear filters. The first problem is that the data is being delayed by the filter which is a feature of linear filters when they are set to have a strong filtering effect. The second issue is that the filtered signal does not seem to follow the original measured data very well. To avoid the delay and provide results of high quality, we can employ an instance of median filter to smooth the measured data.

A relatively small number of samples are sufficient to be able to filter the measured data and remove the outliers. To demonstrate the benefits of the discovered approximations, we will apply the evolved 11-input and 13-input medians to filter the outliers presented in a signal captured by an accelerometer sensor. The obtained non-functional parameters are summarized in Tables 3 and 4. As the non-functional parameters are manually evaluated on real systems, only some of the Pareto dominant discovered solutions are investigated. It should be noted that only the number of operations and the quality defined by Eq. 1 was considered during construction of the Pareto set. We have implemented and measured not only the evolved solutions, but also three common approaches to determine median value—the *quicksort* algorithm, *quickselect* algorithm and the so called *running median*. While *quicksort* represents a sorting algorithm, the *quickselect* is a selection algorithm which is able to find the kth smallest element in an unordered list [4]. The quickselect uses the same overall approach as quicksort, however, it only recurses into one side of the input sequence which reduces the average complexity. The *running median* attempts to minimize processing time by maintaining a data list that is sorted from the smallest value to the largest value [25]. When a new sample is submitted, it replaces the oldest sample. The new sample is then shifted in the sorted list to bring it to the correct location.

Firstly, let us discuss size of the machine code of the complied C codes. If we compare the amount of bytes occupied by median networks and common approaches such as quicksort, quickselect and running median, we can easily

Table 3 Non-functional parameters of accurate (emphasized) and approximated implementations of 11-input median function measured on different MCUs

Impl.	Machine code size [B]				Execution time [μs]				Consumed energy [nWs]			
	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430
14-ops	118	324	328	156	4.0	88	341	310	120	604	682	249
20-ops	158	441	452	224	4.6	108	450	356	140	745	900	286
25-ops	206	549	567	276	5.5	127	552	375	169	878	1105	301
30-ops	232	648	684	318	5.9	144	659	410	179	995	1318	329
32-ops	254	696	845	342	6.2	153	791	420	188	1057	1582	337
38-ops	294	819	1065	400	6.8	175	982	450	207	1208	1964	361
44-ops	328	900	1200	434	7.5	187	1105	465	230	1290	2210	373
50-ops	378	1032	1320	472	8.6	210	1220	480	261	1449	2440	385
qsort	128	333	–	196	40.5	958	–	1515	1235	6610	–	1217
qselect	212	849	607	276	17.5	488	2910	705	535	3367	5820	566
running	236	729	412	344	14.2	274	785	690	435	1887	1570	554

An implementation labelled as *n*-ops denotes evolved comparator network consisting of *n* operations

Table 4 Non-functional parameters of accurate (emphasized) and approximated implementations of 13-input median function measured on different MCUs

Impl.	Machine code size [B]				Execution time [μs]				Consumed energy [nWs]			
	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430
17-ops	138	378	387	192	4.4	96	375	335	135	666	750	269
26-ops	214	588	594	288	5.6	136	587	390	172	935	1174	313
34-ops	288	747	922	402	7.0	163	880	470	215	1125	1760	377
38-ops	330	825	1054	434	8.0	176	980	480	244	1218	1960	385
41-ops	332	894	1144	516	8.0	190	1058	530	245	1309	2115	426
48-ops	398	1020	1306	574	8.8	210	1205	565	270	1452	2410	454
58-ops	478	1209	1590	642	9.5	242	1690	585	290	1672	3380	470
66-ops	496	1353	1854	666	10.2	266	1690	560	311	1835	3380	450
qsort	128	333	–	196	51.6	1178	–	1800	1574	8128	–	1445
qselect	212	849	607	276	21.4	610	4060	800	652	4212	8120	642
running	236	732	394	344	13.1	552	1100	750	400	3809	2200	602

determine that these implementations are more compact compared to the accurate median filter implemented using the median network consisting of 50 min/max operations for the 11-input median and 66 operations for the 13-input median. The size of the quicksort routine is equal to the size of the 11-input approximate median consisting of 14 operations. To sum up, quicksort is the most compact algorithm. Nevertheless, it is interesting to note that PIC16 does not allow one to execute the quicksort algorithm because its implementation relies on the recursion which cannot fit the in-memory emulated stack. The implementation of the running median occupies approximately a 1.8 times higher number of bytes on average compared to the quicksort. Quickselect consumes a bit more except for STM32 and MSP430 where the algorithm requires a lower number of bytes to be implemented.

The number of operations of the approximate median functions correlates with the machine code size. There is only one exception. The 13-input comparator network consisting of 38 operations implemented on STM32 exhibits nearly no reduction compared to the code consisting of 41 operations. It seems that some optimization tricks were discovered by the ARM compiler.

If we compare the size of machine code across all considered microcontrollers, the ARM architecture has an extremely efficient mechanism of instruction encoding. In addition, it revealed that the ARM compiler contains a very effective optimization engine. Similarly, the code generated by the MSP430 GCC compiler is very compact compared to the code for PIC microcontrollers. It is worth noting that it is extremely important to enable GCC optimizations. Otherwise, not only the size of the machine code, but also computation time increases by 60 % on average without changing a line of C code. When we take into account the fact that MSP430 is equipped with an exceptionally large FLASH memory (see Sect. 8.1), it seems to be a very powerful low-cost microcontroller.

The average execution time and average energy consumption measured for various implementations of 11-input and 13-input accurate as well as approximate median functions are given in the second and third part of Tables 3 and 4. Let us first discuss the parameters of the accurate implementations. During the measurements, it turned out that the energy consumption pattern remains almost invariant because all approximations use identical sequences of instructions. Consumed energy thus mainly depends on the execution time which is shorter when more aggressive approximations are applied. The average power consumption, when an accurate median is calculated, is 0.8 mW for MSP430, 2 mW for PIC16, 6.9 mW for PIC24 and 30.5 mW for ARM. In the case of the 11-input median function implemented on PIC24, the median network is 4.5 times faster than the quicksort algorithm and the consumed energy was reduced from 6610 to 1449 nWs (i.e. by 78 %). The median network is 4.7 times faster than quicksort on the STM32 and the energy was also reduced by 78 %. A little bit worse situation is at MSP430. The median network is 3.1 times faster than quicksort, but its energy consumption decreases by 68 %. Similar results were obtained for the 13-input median. The median network implemented on PIC24 is 4.4 times faster than the quicksort algorithm and the consumed energy was reduced by 77 %. At STM32, the quicksort algorithm exhibits 5 times worse execution time and an 80 % higher energy consumption compared to the median calculated using 66 min/max operations. At

MSP430, the median network is executed 3.2 times faster than quicksort. While there is a relative large performance gain of median networks compared to the quicksort algorithm, the execution time of running median is comparable with the median networks. The best improvement is achieved at STM32 where the implementations of median networks are executed 1.7 times faster than running median. For the rest, the gain varies around 1.4 on average.

Let us now move on to the execution time and energy consumption of the evolved approximate median functions. At first glance, it is evident that the execution time decreases with the decreasing number of operations. The situation is, however, a little bit complicated here. Let us compare the execution time of, for example, an 11-input accurate median network consisting of 50 operations and an 11-input reduced network consisting of 25 operations. While the number of operations is reduced by 50 %, the execution time is adequately decreased only at PIC16, where a 54 % improvement was achieved. STM32 and PIC microcontrollers exhibit improvement which is less than 39 %. In the case of MSP430, only a 22 % reduction was achieved. In order to better understand this phenomenon, we have to firstly investigate the dependence between the number of min/max operations and the number of generated instructions for MSP430. The implementation of an accurate median network consists of 219 instructions and the reduced median network consists of 122 instructions which leads to a 44 % improvement. Unfortunately, the difference between the improvement at the level of instructions (44 %) and improvement at the level of operations (50 %) is relatively small. In order to determine the root source of such a discrepancy, it is necessary to perform an analysis at the level of a machine code. It has been revealed that two different mechanisms were used to implement the min/max operations. Some operations are implemented using indirect addressing, other operations are optimized and consists of instructions only manipulated with registers. This makes a huge difference in the number of clock cycles required to execute a single min/max operation. Some operations are evaluated within 5 cycles, others require up to 11 clock cycles. Despite this finding, there is linear dependence between the energy consumption and time of execution and longer times imply a higher energy demand.

Despite the fact that STM32 has the largest current consumption in active mode, it provides the best results from the perspective of energy consumption. Even if the MSP430 is declared as an ultralow-power microcontroller, it requires about a 1.7 times higher amount of energy to execute the same code. It is necessary to note, however, that higher energy consumption is in close relation with the time of execution which is more than 60 times higher compared to STM32. Compared to PIC16 and PIC24, MSP430 consumes from 3 to 6 times lower energy to accomplish the same task. On the other hand, PIC24 is able to produce about five times more results within the same period of time at the cost of 30 % higher energy consumption compared to MSP430. In order to avoid misinterpretation, it is worth noting that MSP430 operates at 1 MHz while PIC24 operates at 4 MHz. When we increase the frequency to 4 MHz, the time of execution decreases four times with no additional cost (the energy consumption remains at the same level).

Figure 6 gives an example of real data filtered by various implementations of 11-input and 13-input median filters. The data were obtained from an accelerometer

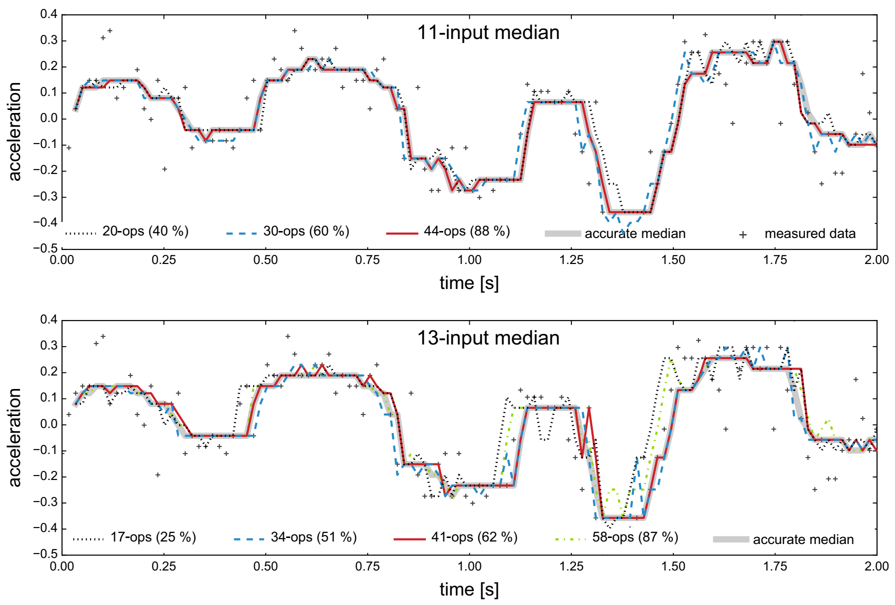

Fig. 6 Example of data filtered using accurate as well as approximate versions of 11-input and 13-input median filter. Note that only some of the *measured points* are shown because of readability

whose output signal was sampled at 8 kHz. Taking into account the sample rate, the considered accurate median filters introduce a delay not worse than 1.7 ms which represents a reasonable value. When six operations (12 %) are removed from the 11-input median network, the resulting approximate median produces an output that is nearly similar to the output of accurate implementation. There are only neglible differences that do not prevent us from applying this inaccurate implementation in an embedded application to filter the outliers and save energy. In the case of implementation at STM32, for example, we can reduce the consumed energy by 11.8 % by introducing the approximated median network consisting of 44 operations.

Interestingly, the median network which consists of 20 operations (60 %) produces a signal which is very similar to the output of an accurate median, despite the fact that the measured signal is very noisy. It seems that the output is of a better visual quality compared to the output of a network having 30 operations. In contrast to the output of an accurate median, there are some small oscillations around 1.7 seconds caused by the oscillations in a signal coming from the accelerometer. Nevertheless, the trend in data is reliably followed. In case these small differences do not represent a real problem for a target application, it is worth implementing the improved median network which is able to offer a 40 % reduction of power consumption on the one hand, its approximately 1.8 times faster execution time on the other hand.

The approximate versions of the 13-input median also performs very well. Only small differences are observable when a median network with 38 % removed operations is used. Compared to a commonly used running median, we obtain a solution which has 38 % lower power consumption when implemented on a STM32 microcontroller. Interestingly, the approximate median networks which consists of 17 and 58 operations exhibit lower delay compared to the fully working 13-input median. It can be seen that the filtered data appears to be shifted to the left when these filters are used.

It can be concluded that the observations on real examples are consistent with conclusions given in Sect. 7. As the quality of an approximate median defined by Eq. 1 decreases, the amount of inaccuracies in the output signal increases. The processing of the sensor data seems to be an application with great potential for genetic improvement. As was previously shown, we are able to significantly improve energy consumed by the filters for a cost of small differences in the output data. In fact, any of the presented approximations can be used to filter the input signal because no golden solution is available for the validation of the obtained outputs. The filtration is typically used to avoid high variances in output signal (i.e. to reduce sensitivity of the output signal to the outliers). In this sense, we can employ approximate medians consisting of 50 % (or even less) operations to accomplish this task because they are able to sufficiently remove the outliers.

8.4 Median in image processing

The processes of acquiring, transmitting and storing images in computer systems are not always ideal and hence some pixels or groups of pixels can be corrupted. Hence, noise elimination is a typical low level image processing task. In many applications, the noise elimination has to be implemented by non-linear functions because the noise contained in the images is inherently non-linear [6]. A typical representative of non-linear noise is a shot noise which manifests itself by setting some individual pixels to a random value. Median-based non-linear filters play a prominent role among the filters utilized to suppress the shot noise [2]. Traditionally, a simple median filter applied to every pixel of the input image is employed. In advanced image filters (e.g. switching filters [32]) the filtering function is only applied if a noise detector, implemented typically using a median, detects some noise.

The image filters operate with pixel values in the neighbourhood of the centre pixel. The process of filtration is based on a sliding window, a square window of an odd size $(2k + 1)$, that moves along the image. More formally, let I be an image consisting of $m \times n$ pixels $x(i,j) \in I$, where $1 \leq i \leq m, 1 \leq j \leq n$. Then, each pixel of the filtered image I' is calculated as $y(m,n) = \text{median}(W_I(m,n))$, where $W_I(m,n) = \{x(m+i, n+j) \in I \mid -k < i,j < k\}$ is a sliding window function. It is evident that the median value is calculated using $(2k+1)^2$ pixels. The typical sliding windows employed in image processing consists of 3×3 and 5×5 pixels which corresponds with 9-input and 25-input filtering functions.

The measured non-functional parameters of various implementations of 9-input accurate as well as approximate median filters are summarized in Table 5. Apart from the evolved implementations, two common approaches to determine a median value are evaluated—the quicksort algorithm and the quickselect algorithm. The

Table 5 Non-functional parameters of accurate (emphasized) and approximated implementations of 9-input median function measured on different MCUs

Impl.	Machine code size [B]			Execution time [µs]			Consumed energy [nWs]		
	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430

Impl.	Machine code size [B]				Execution time [µs]				Consumed energy [nWs]			
	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430	STM32	PIC24	PIC16	TI430
9-ops	78	204	207	96	3.2	65	228	274	97	450	457	220
10-ops	84	234	238	108	3.3	71	256	280	102	492	512	225
14-ops	112	315	324	156	3.9	86	338	310	118	590	675	249
16-ops	126	372	376	176	4.1	96	386	324	126	666	771	260
20-ops	158	441	454	208	4.6	108	452	340	141	745	905	273
22-ops	180	495	502	234	5.0	118	506	360	151	818	1012	289
26-ops	208	573	586	280	5.4	132	576	388	165	909	1153	312
30-ops	240	645	676	330	6.4	144	650	412	196	994	1299	331
qsort	128	333	–	196	26.8	830	–	1325	816	5727	–	1064
qselect	212	849	607	272	15.3	466	2255	690	467	3219	4510	554

running median discussed in the previous section is not applicable in this case because more than one value has to be removed/added between two subsequent processing windows. The discussion that has been given for the implementation of the 11-input median and its variants is also valid for the 9-input alternative whose parameters are included in Table 5. There is nearly a linear dependency between the number of operations used to approximate the median value and the execution time as well as power consumption.

The results for the 25-input median and its alternative implementations are given in Table 6. In contrast with the 13-input approximate medians, the difference between the improvement achieved at the level of operations and improvement at the level of instructions does not exceed 5 %. Similarly, the time of execution decreases linearly with a decreasing number of operations with one exception—implementation compiled for MSP430 which suffers from issues observed also for 13-input approximate medians. There is an 18 % difference between the reduction at the level of instructions and the reduction of execution time (see the execution time for 174-ops and 60-ops implementations). Since the response of other architectures to a reduced number of operations is as expected, it may suggest that there may be a problem with the quality of the compiled code. We did not analyse this problem in detail as it is outside the scope of this paper.

The chosen problem nicely demonstrates the overhead of median networks implemented in the software. The accurate median function implemented using 174 operations occupies ten times more bytes than the quicksort algorithm. Even if we remove half of the operations, the machine code is more than six times larger. This is the price that must be sacrificed for greater speed of the algorithm based on a median network. As regards the execution time, the median can be calculated 70 % faster when the median network which consists of 174 operations is used instead of the quicksort algorithm and 31 % faster when compared to the quickselect

Table 6 Non-functional parameters of accurate (emphasized) and approximated implementations of 25-input median function measured on different MCUs

Impl.	Machine code size [B]			Execution time [μs]			Consumed energy [nWs]		
	STM32	PIC24	TI430	STM32	PIC24	TI430	STM32	PIC24	TI430
60-ops	502	1302	742	10.9	262	665	333	1808	534
70-ops	596	1527	912	12.3	303	785	375	2091	630
88-ops	796	1887	1180	16.4	366	955	501	2525	767
107-ops	920	2250	1438	18.4	428	1100	562	2953	883
150-ops	1264	3015	1688	23.9	554	1130	727	3823	907
160-ops	1378	3195	1818	24.6	584	1200	751	4030	964
164-ops	1454	3255	1826	26.0	596	1240	793	4109	996
174-ops	1524	3423	1864	27.6	619	1270	841	4271	1020
qsort	128	333	196	104.0	2430	2610	3172	16,767	2096
qselect	212	849	276	39.1	1040	1685	1194	7176	1353

Note that PIC16 is not included in this table due to small amount of available RAM memory algorithm. The 25-median implemented using 150 operations enables us to reduce the energy by more than 10 %. According to the distribution of errors shown in Fig. 5, this implementation provides an output of high quality with a low percentage of erroneous outputs that are close to the median value.

In order to evaluate the filtering quality as well as robustness of the evolved approximate medians, the medians were employed as median filters and evaluated using 25 randomly selected test images (384x256 pixels) from [17] that were corrupted by random valued shot noise. Because the removal of random valued shot noise represents a difficult problem, it usually is used to compare the performance of various median filters [5]. Considering the fact that a sliding window is used, more than two million test cases were in fact used for quality assessment. There exists several approaches to measure the quality of filtered images. The structural similarity index (SSIM) represents probably the most advanced approach which attempts to quantify the visibility of errors (differences) between a distorted image and a reference image[37].

Boxplots of the structural similarity index calculated for 9-input and 25-input accurate as well as approximate median networks used as image filters are given in Fig. 7. As is evident from the results, there is a relatively large variance in the similarity index of accurate as well as inaccurate median filters. The index of similarity for images produced by accurate an 9-input median filter is 88.6 % in average. The average similarity index decreases with the decreasing number of operations. Interestingly, it decreases very slightly without any radical change in variance. When we reduce the number of operation to 16 (53 %), the average similarity index decreases to 87.5 %. The results suggest that it is possible to use an approximate median network consisting of half the number of operations instead of an accurate median. The impact on quality of the filtered images is negligible.

Figure 8 illustrates filtering capabilities of various filters on an image corrupted by random valued shot noise where 10 % of the pixels are affected. The output of the median filter and approximate median filter is visually indistinguishable. Nearly all of the noisy pixels were successfully detected and removed, even for a median with 16 operations. When we reduce the number of operations to 14, a few noisy pixels remain in the filtered image. This behaviour corresponds with the distribution

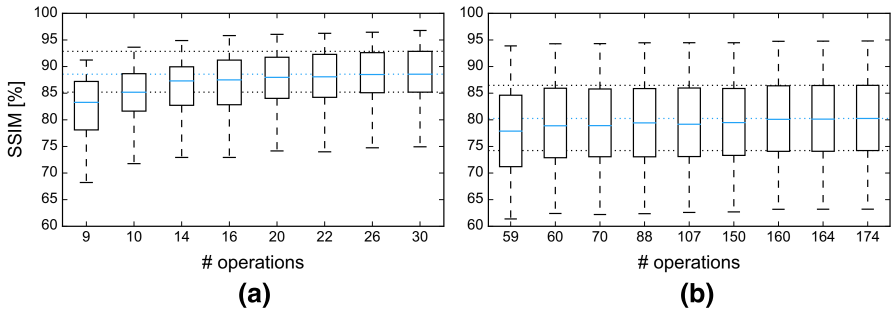

Fig. 7 *Boxplots* illustrating the distribution of structural similarity index for evolved median networks calculated using a set of test images corrupted by 10 % random valued shot noise. **a** 9-input median. **b** 25-Input median

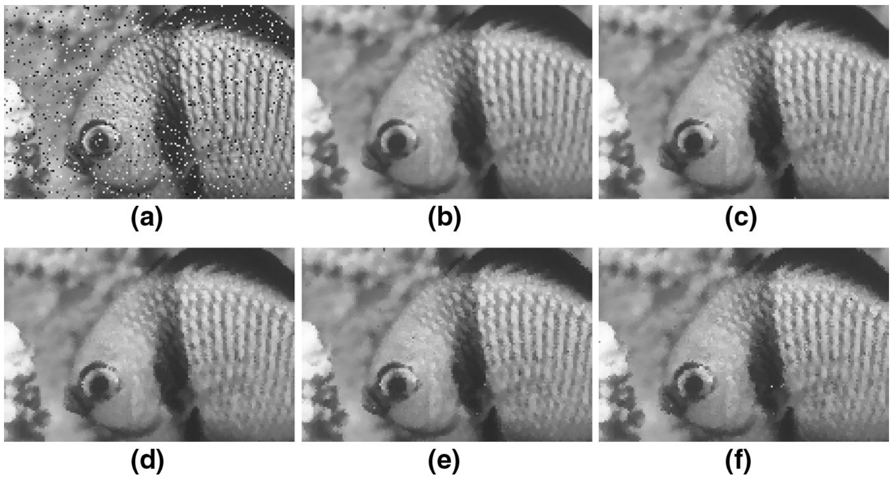

Fig. 8 Detail of an image **a** corrupted by 10 % random valued shot noise filtered by **b** 9-input accurate median filter and approximated median filters consisting of **c** 22 (73 %) operations, **d** 16 (53 %), **e** 14 (46 %) and **f** 10 (33 %) operations

of errors shown in Fig. 5 and a detailed analysis provided in Table 2. The 9-input approximate median with 16 operations exhibits the worst-case distance error equal to one, while the 14-ops implementation has the worst-case distance error equal to two.

If we compare the distribution of the similarity index for a 9-input and 25-input median filter, it is evident that the 25-input median filters provide results of lower quality. The similarity index of the accurate implementation consisting of 174 operations is equal to 80.3 %. The reason is obvious. Increasing the size of the filtering window allows for the common median filter to remove a great deal of noisy pixels, however, because the standard median filters modify almost all pixels, images become smudged and less detailed. Nevertheless, this fact does not prevent the employment of the 25-input median filter as a robust noise detector. Interestingly, there is only a small degradation in quality of the reduced 25-input median filters. When we remove 50 % of operations, the similarity index decreases to 79.4 % on average. This approximation yields a 40 % reduction in power consumption when implemented on STM32 microcontroller.

Similar conclusions may be inferred even if we use the peak signal-to-noise ratio (PSNR) which represents another commonly used quality metric. In contrast to structural similarity, PSNR does not respect a psycho-visual model of the human optical system. While PSNR of the images filtered by the accurate 9-input median filter is equal to 29.4 dB in average, PSNR of the images obtained by the 14-ops (9-ops) filter drops by 1.3 dB (3.5 dB). The PSNR of the images filtered by the accurate 25-input median filter is equal to 25.9 dB. When the number of operations is reduced to 59, the PSNR only decreases by 0.7 dB.

The results demonstrate how robust the evolved implementations are and that there is great space for improvement of the non-functional parameters in practice. In most cases, it is not even necessary to exactly determine the median value which helps us to reduce the power consumption or increase the performance (i.e. speed) of a given piece of software.

9 Conclusions

In this paper, we presented a new approach to improve non-functional properties of software. In particular, we concentrated on improvements in the execution time and power consumption of various instances of the median function. In general, it is impossible to improve non-functional parameters of the median function without accepting occasional errors in results since optimal implementations of typical instances are available. In order to address this problem, we adopted the approximate computing scenario which allows us to accept some errors in the outputs.

In approximate computing, software and hardware is approximated, i.e. simplified with respect to fully accurate implementations, in order to reduce power consumption or increase performance. As a consequence, errors can emerge during computations which is tolerable in many real applications. When an approximation should be introduced, the common approach is to remove the less significant bits and reduce data widths. This paper shows that the approximation conducted at the level of function (algorithm) that are based on EA is able to deliver significantly better results.

The median is implemented using a sequence of elementary operations that forms a median network. The task is formulated as a single objective optimization problem where the number of operations represents constraints specified by the designer. The constrains-oriented approach is relevant to practice where the designers usually wants to achieve a particular power reduction in order to improve the performance of the whole embedded system. The method is based on cartesian GP and exploits the fact that GP is able to find a good trade-off between the error and number of operations, even if the number of operations is intentionally constrained.

In order to avoid problem with seeding (only fully functional implementations of various instances of median filter exist), we proposed to apply a two-stage procedure. The first stage starts with a fully functional median network and gradually reduces the number of employed operations in order to satisfy constraints given by a designer. As soon as a satisfactory candidate solution is found, the second stage responsible for maximizing the quality of partially working implementations is used.

The accuracy of determining a median value is measured by means of a problem-specific quality metric. The proposed metric is based on the positional error calculated using the permutation principle introduced in this paper. The impact of the permutation principle was discussed from a theoretical as well as a practical point of view. Firstly, the permutation principle helps us to reduce the computational complexity of the fitness evaluation. Secondly, the permutation principle enables one to construct a metric approaching the quality of selecting the median value and, what is important, which can be efficiently calculated. Finally, the permutation principle helps to understand how to avoid biased solutions that may be produced when we generate test vectors used to determine the fitness score inappropriately (randomly). In order to understand this phenomenon, it is necessary to realize that the median value is determined according to a set of values (i.e. the ordering of input values is completely ignored). It was illustrated and discussed that it is necessary to generate test vectors from different equivalence classes so as to avoid any bias.

The problem of trading between quality and non-functional parameters was demonstrated in four different instances of the median function that are typically employed in practice. The performance of the best discovered approximated median filters was evaluated in two real-world problems—sensor data processing and image processing. The non-functional parameters were measured for four microcontrollers so as to avoid misleading conclusions. The results confirmed that median functions are very good examples of functions for which it makes sense to introduce their approximate versions. When the approximate medians are employed in a particular application, the output quality remains relatively high, even for significant reductions of the number of operations. Hence significant improvements in energy consumption can be obtained.

Even though the permutation principle as well as the proposed error metric are problem specific, this paper demonstrated the ability of GI to provide competitive solutions for a chosen real-world problem from the area of approximate computing. This opens a complete new application area for GI. The ability to deliver partially working solutions seems to be natural for evolutionary techniques. Hence the approximate computing seems to have a great potential for these techniques.

There are several directions for future research. Execution time and power consumption are two possible non-functional criteria that can be optimized. There are additional criteria such as delay that need to be considered, especially if median networks would be implemented in the hardware. Despite the fact that the proposed permutation principle helps to significantly improve the time required to determine the fitness value, the test based approach used to calculate the fitness score represents a bottleneck of the whole framework. Unfortunately, this is a general problem of all generate-and-test-based evolutionary approaches. As a consequence of that, only a subset of all possible permutations was used for quality assessment. This simplification introduces two issues. Firstly, it means that we are not able to guarantee the worst-case error unless all the input permutations are tested. Secondly, it may happen that the quality of a given network is worse than determined. Suprisingly, the experiments revealed that our simplification does not have any significant effect in practice. We are convinced, however, that these issues can be completely eliminated by introducing a formal method based on BDDs to the fitness function.

Acknowledgments This work was supported by the Czech science foundation project 14-04197S—Advanced Methods for Evolutionary Design of Complex Digital Circuits.

References

1. A. Agapitos, S.M. Lucas, Evolving efficient recursive sorting algorithms, in *IEEE Congress on Evolutionary Computation*, pp. 2677–2684 (2006)
2. R.H. Chan, C.W. Ho, M. Nikolova, Salt-and-pepper noise removal by median-type noise detectors and edge-preserving regularization. IEEE Trans. Image Process. **14**, 1479–1485 (2005)
3. B. Cody-Kenny, E.G. Lopez, S. Barrett, locoGP: improving performance by genetic programming java source code, in *Genetic Improvement 2015 Workshop*, ed. by W.B. Langdon, J. Petke, D.R. White (ACM, Madrid, 2015), pp. 811–818
4. N. Devillard, *Fast Median Search: An ANSI C Implementation* (1998). http://ndevilla.free.fr/median/median.pdf
5. Y. Dong, A new directional weighted median filter for removal of random-valued impulse noise. IEEE Signal Process. Lett. **14**(3), 193–196 (2007)
6. E.R. Dougherty, J.T. Astola, (eds.) *Nonlinear Filters for Image Processing. SPIE/IEEE Series on Imaging Science and Engineering*. SPIE/IEEE (1999)
7. H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, Neural acceleration for general-purpose approximate programs. Commun. ACM **58**(1), 105–115 (2014)
8. B.W. Goldman, W.F. Punch, Analysis of cartesian genetic programming's evolutionary mechanisms. IEEE Trans. Evol. Comput. **19**(3), 359–373 (2015)
9. J. Han, M. Orshansky, Approximate computing: An emerging paradigm for energy-efficient design, in *Proceedings of the 18th IEEE European Test Symposium*, pp. 1–6. IEEE (2013)
10. M. Harman, B.J. Jones, Search-based software engineering. Inf. Softw. Technol. **43**, 833–839 (2001)
11. W.D. Hillis, Co-evolving parasites improve simulated evolution as an optimization procedure. Phys. D **42**(1–3), 228–234 (1990)
12. H. Juille, Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces, in *Genetic Algorithms: Proceedings of the 6th International Conference (ICGA95)*, ed. by L. Eshelman (Morgan Kaufmann, Pittsburgh, PA, USA, 1995), pp. 351–358
13. R.E. Kalman, A new approach to linear filtering and prediction problems. Trans. ASME J. Basic Eng. **82**(Series D), 35–45 (1960)
14. D.E. Knuth, *The Art of Computer Programming*, vol. 3, 2nd edn. (Sorting and Searching. Addison Wesley Longman Publishing Co., Inc, Redwood City, 1998)
15. W.B. Langdon, M. Harman, Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. **19**(1), 118–135 (2015)
16. R. Maronna, D. Martin, V. Yohai, *Robust Statistics: Theory and Methods, Wiley Series in Probability and Statistics* (Wiley, New Jersey, 2006)
17. D. Martin, C. Fowlkes, D. Tal, J. Malik, A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics, in *Proceedings of the 8th International Conference Computer Vision*, vol. 2, pp. 416–423 (2001)
18. J.F. Miller, *Cartesian Genetic Programming* (Springer, Berlin, 2011)
19. J.F. Miller, S.L. Smith, Redundancy and computational efficiency in cartesian genetic programming. IEEE Trans. Evol. Comput. **10**(2), 167–174 (2006)
20. V. Mrazek, Z. Vasicek, L. Sekanina, Evolutionary approximation of software for embedded systems: Median function, in *Genetic Improvement 2015 Workshop*, ed. by W.B. Langdon, J. Petke, D.R. White (ACM, Madrid, 2015), pp. 795–801
21. K. Nepal, Y. Li, R.I. Bahar, S. Reda, Abacus: A technique for automated behavioral synthesis of approximate computing circuits, in *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '14*, pp. 1–6. EDA Consortium (2014)
22. J. Petke, M. Harman, W.B. Langdon, W. Weimer, Using genetic improvement and code transplants to specialise a C++ program to a problem class, in *17th European Conference on Genetic Programming, LNCS*, vol. 8599, ed. by Miguel Nicolau, et al. (Springer, Granada, Spain, 2014), pp. 137–149
23. R. Poli, W.B. Langdon, N.F. McPhee, *A Field Guide to Genetic Programming*.Published via http://lulu.com and http://www.gp-field-guide.org.uk (2008)
24. A. Sampson, W. Dietl, E. Fortuna, Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: Approximate data types for safe and general low-power computation, in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 164–174. ACM (2011)

25. P. Schmidt, Simple median filter library designed for the arduino platform (2014). https://github.com/daPhoosa/MedianFilter
26. E. Schulte, J. Dorn, S. Harding, S. Forrest, W. Weimer, Post-compiler software optimization for reducing energy, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14 (ACM, Salt Lake City, 2014), pp. 639–652
27. L. Sekanina, Evolutionary design space exploration for median circuits, in *Applications of Evolutionary Computing, LNCS 3005*, pp. 240–249. Springer (2004)
28. L. Sekanina, M. Bidlo, Evolutionary design of arbitrarily large sorting networks using development. Genet. Progr. Evolv. Mach. **6**(3), 319–347 (2005)
29. L. Sekanina, Z. Vasicek, Approximate circuits by means of evolvable hardware. in *Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI), 2013 IEEE International Conference on Evolvable Systems*, pp. 21–28. IEEE CIS (2013)
30. P. Sitthi-Amorn, N. Modly, W. Weimer, J. Lawrence, Genetic programming for shader simplification. ACM Trans. Gr. **30**(6), 152:1–152:12 (2011)
31. J.L. Smith, Implementing median filters in xc4000e fpgas. XCell **23**(1), 16 (1996)
32. T. Sun, Y. Neuvo, Detail-preserving median based filters in image processing. Pattern Recognit. Lett. **16**, 341–347 (1994)
33. V.K. Valsalam, R. Miikkulainen, Using symmetry and evolutionary search to minimize sorting networks. J. Mach. Learn. Res. **14**(1), 303–331 (2013)
34. Z. Vasicek, L. Sekanina, Evolutionary approach to approximate digital circuits design. IEEE Trans. Evol. Comput. **19**(3), 432–444 (2015)
35. Z. Vasicek, K. Slany, Efficient phenotype evaluation in cartesian genetic programming, in *Proceedings of the 15th European Conference on Genetic Programming, LNCS 7244*, pp. 266–278. Springer Verlag (2012)
36. S. Venkataramani, A. Sabne, V.J. Kozhikkottu, K. Roy, A. Raghunathan, Salsa: systematic logic synthesis of approximate circuits, in *The 49th Annual Design Automation Conference 2012*, DAC '12, pp. 796–801. ACM (2012)
37. Z. Wang, A. Bovik, H. Sheikh, E. Simoncelli, Image quality assessment: from error visibility to structural similarity. IEEE Trans. Image Process. **13**(4), 600–612 (2004)
38. D.R. White, A. Arcuri, A. John, Evolutionary improvement of programs. IEEE Trans. Evol. Comput. **15**(4), 515–538 (2011)
39. A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmailzadeh, K. Bazargan, Axilog: Language support for approximate hardware design, in *Design, Automation and Test in Europe, DATE'15*, pp. 1–6. EDA Consortium (2015)