Evolutionary Approximation of Software for Embedded Systems: Median Function

Vojtech Mrazek
Faculty of Information Technology
Brno University of Technology, Czech Republic
imrazek@fit.vutbr.cz

Zdenek Vasicek
Faculty of Information Technology
Brno University of Technology, Czech Republic
vasicek@fit.vutbr.cz

Lukas Sekanina
Faculty of Information Technology
Brno University of Technology, Czech Republic
sekanina@fit.vutbr.cz

ABSTRACT

This paper deals with genetic programming-based improvement of non-functional properties of programs intended for low-cost microcontrollers. As the objective is to significantly reduce power consumption and execution time, the approximate computing scenario is considered in which occasional errors in results are acceptable. The method is based on Cartesian genetic programming and evaluated in the task of approximation of 9-input and 25-input median function. Resulting approximations show a significant improvement in the execution time and power consumption with respect to the accurate median function while the observed errors are moderate.

Categories and Subject Descriptors

I.2.8 [**Computing methodologies**]: Artificial intelligence—*Problem Solving, Control Methods, and Search*;
D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*

Keywords

Genetic Improvement; Genetic Programming; Cartesian Genetic Programming; Approximate Computing; Embedded Systems

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 [10]. 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 [4]. The aim of genetic improvement, conducted by genetic programming, is to improve non-functional parameters of programs [16], and even improve existing software [8].

GECCO '15, July 11 - 15, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739482.2768416

A similar research direction has been explored in the field of approximate computing which is a promising approach to obtain energy-efficient computer systems. 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, power consumption and performance. The approximations can be introduced at the level of hardware as well as software. The process of approximation is usually based on a heuristic procedure. For instance, software implementing functions such as edge detection, FFT etc. were approximated by artificial neural networks (ANN) in order to accelerate computations and reduced power consumption (when the ANN is implemented on a chip) [2]. Using genetic programming in the context of approximate computing has been reported for digital circuits approximation [14, 15].

In this paper, we deal with GP-based improving of non-functional properties of programs that are intended for low-cost microcontrollers. As we seek for 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 this paper is to find programs showing suitable compromises between the accuracy, execution time and power consumption for 9- and 25-input median functions (9-median and 25-median for short) when implemented on a microcontroller. The number of inputs corresponds with a typical size of processing window (3×3 and 5×5 pixels) employed in image processing. The approximations will be performed by Cartesian genetic programming and evaluated using data sets and by physical measurements on three microcontrollers: the 8-bit microcontroller of Microchip PIC family with code name PIC16F628A, 16-bit microcontroller PIC24F08KA102 and 32-bit ARM-based microcontroller STM32F100RB.

Section 2 surveys relevant research. The CGP-based approximation is described in Section 3. Section 4 provides a detailed analysis of evolved approximate implementations of 9-median and 25-median in terms of accuracy, execution time and energy consumption. Conclusions are given in Section 5.

2. RELEVANT WORK

The state of the art section is decomposed into three parts dealing with (1) the calculation of the median, (2) principles of genetic improvement, and (3) approximate computing.

2.1 Determining the median

Given a finite sequence of data samples, 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, as it is the statistic that is the most resistant to outliers that could be presented in a given sequence. This feature is widely exploited in the 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. The straightforward approach is to employ a generic sorting algorithm, for example the most popular and efficient quicksort algorithm. This algorithm is very compact and robust, however, the number of steps needed to determine the median value depends on 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.

The alternative way of calculating the median value is to use a median network. The median network is a kind of sorting networks whose concept is deeply elaborated in [6]. Sorting network is defined as a network of elementary operations denoted as compare&swap (CS) elements that sorts all input sequences. The sorting network can be constructed using a sorting algorithm which must be data independent. Bitonic-sorting and Batcher's odd-even merge sorting are examples of such algorithms. A compare&swap of two elements (a, b) compares a and b and exchanges (if it is necessary) the elements in order to obtain sorted sequence. The CS can be implemented using two operations – minimum and maximum. The advantage of median network is that the sequence of CS operations depends only on the number of input elements, not on the values of the elements. In addition to that, the total number of operations is in practice lower compared to the number of operations that have to be executed when a common sorting algorithm is used.

2.2 Genetic improvement

Genetic programming is a method allowing automated design and optimization of programs and other entities that can be represented and simulated in a computer [7]. In the context of this work, one can observe that evolutionary design and/or optimization of a (accurate) median outputting program was carried out by GP only rarely [12]. However, considerable amount of research papers were devoted to the design and optimization of sorting algorithms (e.g., [1, 16]) and sorting networks (e.g. [5, 13]), which are useful structures when the median value has to be obtained. As checking whether a specification (or an original code) and a candidate solution are semantically equivalent is time consuming, the 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 using other data sets. A genetic improvement of sorting algorithms was demonstrated in [16], where GP enabled to discover code optimization tricks probably unreachable by current compilers. Evolved code was evaluated using the M5 Simulator targeted for an Alpha processor. While the previous example has dealt with non-functional improvements, Langdon and Harman showed that GP can, in addition to non-functional parameters, improve functionality of existing code [8]. Such improvements can be expected in software which is processing large volumes of data using various heuristic procedures and trying to minimize an error metric.

2.3 Approximate computing

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 emerge during computations. In many cases the 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.

There are many examples of manual approximations in the literature [3]. As manual approximation is not an efficient design method, systematic methods have been developed, e.g. [9]. In our previous work, we used CGP to approximate digital circuits such as adders, multipliers and median-outputting circuits [14, 15]. Artificial neural networks were proposed in [2] to learn to behave like general-purpose code written in an imperative language. The trained network then replaced the original code. A more general approach is EnerJ [11], an extension to Java that adds approximate data types. 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. In addition, the system can statically guarantee isolation of the precise program component from the approximate component. Axilog is a set of language annotations that provide the necessary syntax and semantics for approximate hardware design and reuse in Verilog [17]. Axilog enables the designer to relax the accuracy requirements in certain parts of the design, while keeping the critical parts strictly precise.

3. EVOLUTIONARY APPROXIMATION

In order to approximate 9-median and 25-median, we used a standard CGP according to [15]. In CGP, candidate programs are represented in an array of $n_r \times n_c$ functional nodes. In our case, only two functions (minimum and maximum) are allowed in the function set. As a sequential code (and without loops) has to be evolved, the functional nodes will be arranged in a one-dimensional array, i.e. $n_r = 1$, whose size will be limited by n_c. No feedbacks are allowed. The l-back parameter, determining where the addresses of nodes' operands can be located, will be unrestricted, i.e. $l = n_c$. The number of program inputs is $n_i = 9$ (for 9-median) or $n_i = 25$ (for 25-median). The number of program outputs is $n_o = 1$ (the median value). Each candidate program is thus encoded using $3n_c + 1$ integers.

The search is performed using a $(1 + \lambda)$ strategy, in which λ offspring programs are generated form the parent using the mutation operator. In our case, $\lambda = 4$ and 5% of the chromosome undergoes the mutation. The number of gen-

erations is limited by $g_{max} = 3 \times 10^6$ for the 9-median and $g_{max} = 300 \times 10^3$ for the 25-median which corresponds to 3 hour CGP runs. All the parameter values were set up according to [15]. In order to evaluate a candidate program, we randomly generated 10^4 training vectors for the 9-median and 10^5 vectors for the 25-median and calculated the mean absolute error as the fitness.

The evolutionary approximation exploits the idea that CGP is capable of minimizing the error even if resources (the number of available functional nodes) are not sufficient for obtaining a fully functional solution [14]. The accurate 9-median (25-median, respectively) requires 38 nodes (220 nodes, respectively). Hence the aforementioned CGP-based process was repeated with constrained resources, n_c =6, 10, 14, 18, 22, 26, 30 and 34 nodes for 9-median, and n_c =10, 40, 70, 100, 130, 160, 170 and 200 nodes for 25-median. The best-obtained approximations taken from 50 independent CGP runs are reported in Table 3 and 4.

4. ANALYSIS OF EVOLVED CODE

4.1 Microcontrollers used for testing

Three microcontrollers were chosen to evaluate the parameters of evolved approximate median functions: The 8-bit microcontroller of Microchip PIC family with code name PIC16F628A, 16-bit microcontroller PIC24F08KA102 and 32-bit ARM-based microcontroller STM32F100RB.

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 implemented. 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 run 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. This chip consumes about 4 mA in the active mode and 25 nA in the sleep mode.

The STM32F100RB incorporates a high-performance RISC ARM Cortex M-3 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 the run mode ranges from 10 mA to 150 mA depending on the state of peripherals.

4.2 Evolved code on different microcontrollers

Obtaining a program code from evolved genotype is straightforward. Every active node, starting from one with the lowest index, is represented by an operation (min or max) whose operands are taken from the input sequence or the outputs of preceding operations. In order to implement evolved functions in a microcontroller, we used the C language. Example of a function which approximates 9-median using 18 operations is shown in Listing 1.

Listing 1: Approximation of 9-median using 18 operations

```
dtype median9(dtype* din)
{
        dtype s0=min(din[2],din[3]);
        dtype s1=max(din[5],din[4]);
        dtype s2=max(din[2],din[3]);
        dtype s3=min(din[4],din[5]);
        dtype s4=min(din[0],din[1]);
        dtype s5=max(din[7],din[6]);
        dtype s6=min(din[8],s5);
        dtype s7=max(din[0],din[1]);
        dtype s8=max(s4,s0);
        dtype s9=max(s8,s3);
        dtype s10=min(din[6],din[7]);
        dtype s12=min(s1,s7);
        dtype s13=min(s12,s2);
        dtype s14=max(s6,s9);
        dtype s15=min(s6,s9);
        dtype s16=max(s13,s15);
        dtype s17=max(s10,s16);
        dtype s18=min(s14,s17);
        return  s18;
}
```

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. Note that dtype is chosen to fit the data word width of the target processor, i.e. 8-bit, 16-bit and 32-bit.

To illustrate the relation of the generated machine code complexity and a target platform, we will show fragments of code that are responsible for calculating minimal and maximal value. The code which was used for PIC12 microcontrollers is given in Listing 2. It is expected that the first operand is stored at memory location denoted as VAR_A and the second operand at location VAR_B. The calculated value is stored at memory location VAR_C. To determine the relation between the input operands, subtraction operation is used. This instruction performs the operation $d = f - W$, where f is a location within the RAM memory, W is the accumulator and d can be accumulator or memory location. The given fragment consisting of 6 instructions is optimized to be executed in 6 clock cycles even if the number of cycles required by a branching instruction btfss (skip next instruction if a bit of a register is set) varies between one and two

Listing 2: MIN and MAX using PIC16 assembly

```
MAX:   movf   VAR_B, w     ; w = VAR_B
       subwf  VAR_A, w     ; w = VAR_A - w
       movf   VAR_B, w     ; W = VAR_B
       btfss  status, 0x0  ; skip next if carry set
       movf   VAR_A, w     ; W = VAR_A
       movf   w, VAR_C     ; VAR_C = MAX(VAR_A,VAR_B)

MIN:   movf   VAR_B, W     ; w = VAR_B
       subwf  VAR_A, W     ; W = VAR_A - w
       movf   VAR_A, w     ; W = VAR_A
       btfss  status, 0x0  ; skip next if carry set
       movf   VAR_B, w     ; W = VAR_B
       movf   w, VAR_C     ; VAR_C = MIN(VAR_A,VAR_B)
```

cycles depending on the result of the test. If the conditional test is true, the instruction requires two cycles. Otherwise, one clock cycle is needed. The C code was compiled by the XC8 compiler which is integrated in PIC IDE.

The code generated by the XC16 compiler targeting 16-bit microcontrollers is given in Listing 3. Both fragments expect that the input operands are loaded in registers w4 and w5. The output is stored in register w0 and determined using a subtraction instruction. The instructions of PIC24 are designed in such a way, that each instruction is executed within a single clock cycle. However, if there is a conditional branch and the condition is met, one cycle penalty is introduced. Taking into account this rule, 4 clock cycles are required to determine the minimum (maximum) value using the code shown in Listing 3.

Listing 3: MIN and MAX using PIC24 assembly

```
MAX:    sub    w4, w5, [w15] ; tmp = w4-w5
        mov    w5, w0
        bra    le, max_w5
max_w4: mov    w4, w0
max_w5:


MIN:    sub    w4, w5, [w15] ; tmp = w4-w5
        mov    w5, w0
        bra    ge, min_w5
min_w4: mov    w4, w0
min_w5:
```

The code generated using Atollic ARM compiler is given in Listing 4. The input operands are loaded in registers r2 and r3 and the output is stored in register r0. In contrast with the instruction set of PIC microcontrollers, ARM contains an instruction for comparing two registers. All the instructions are executed within a single clock cycle apart from the conditional branch. This instruction can cause pipeline flush. The number of cycles required for a pipeline refill, however, ranges from 1 to 3 depending on the alignment and width of the target instruction, and whether the processor manages to speculate the address early.

Listing 4: MIN and MAX using ARM assembly

```
MAX:    cmp     r2, r3
        ble.n   max_r3
        mov     r0, r2
        b.n     max_r2
max_r3: mov     r0, r3
max_r2:

MIN:    cmp     r2, r3
        bge.n   min_r3
        mov     r0, r2
        b.n     min_r2
min_r3: mov     r0, r3
min_r2:
```

The size of routines implementing approximate 9-median functions are given in Table 1, where x-ops denotes an implementation utilizing x operations. The size is expressed in the number of bytes as well as the percentage of the total memory capacity available on a given chip.

In the case of PIC16, a stack which could be used to store the temporary values is not available. As a consequence of that, the recursive quicksort algorithm can not be implemented as its implementation relies on the recursion. The size of median routines increases with the increasing number of utilized operations.

If we compare the size of a bytecode for PIC24 and STM32, it can be seen that the ARM compiler produces a more compact code. This is caused by the fact that the instructions are encoded more efficiently.

The quicksort-based implementation is more compact compared to the accurate median filter implemented using a median network. The median network consisting of 28 operations occupies approximately two times higher number of bytes. The size of the quicksort routine is equal to the size of an approximate median consisting of 18 operations.

Table 1: Size of machine code for approximate 9-input median functions

	Target platform		
Impl.	**STM32**	**PIC24**	**PIC16**
6-ops	52 B (0.04%)	144 B (2%)	146 B (7%)
10-ops	80 B (0.06%)	234 B (3%)	234 B (11%)
14-ops	108 B (0.08%)	321 B (4%)	322 B (16%)
18-ops	148 B (0.11%)	417 B (5%)	416 B (20%)
22-ops	176 B (0.13%)	489 B (6%)	484 B (24%)
26-ops	212 B (0.16%)	567 B (7%)	560 B (27%)
30-ops	232 B (0.18%)	639 B (8%)	638 B (31%)
34-ops	252 B (0.19%)	711 B (9%)	872 B (43%)
38-ops	280 B (0.21%)	783 B (10%)	961 B (47%)
qsort	144 B (0.11%)	387 B (5%)	—

The number of bytes required to implement approximate 25-median functions is given in Table 2. Note that PIC24 is not included in this table due to the small amount of RAM memory. Similarly to the previous case, the code size increases linearly with the increasing number of operations. The accurate median network occupies ten times more bytes than the quicksort algorithm. This is the price that must be sacrificed for great speed of the algorithm based on a median network.

Table 2: Size of machine code for approximate 25-input median functions

	Target platform	
Implementation	**STM32**	**PIC24**
10-ops	84 B (0.1%)	237 B (3%)
40-ops	328 B (0.3%)	888 B (11%)
70-ops	640 B (0.5%)	1527 B (19%)
100-ops	904 B (0.7%)	2103 B (26%)
130-ops	1208 B (0.9%)	2643 B (32%)
160-ops	1512 B (1.2%)	3186 B (39%)
170-ops	1532 B (1.2%)	3360 B (41%)
200-ops	1868 B (1.4%)	3897 B (48%)
220-ops	2052 B (1.6%)	4251 B (52%)
qsort	144 B (0.1%)	387 B (5%)

4.3 Accuracy

The quality of approximate software can be evaluated using various metrics, for example, using the error probability

(error rate) which is defined as the percentage of inputs vectors for which the approximate output differs from the original one. However, this commonly applied metric does not reflect the quality of selecting the median value. For example, there can exist an algorithm slightly modifying one half of the output values and still providing good performance if used, for example, in image filtering. Other commonly used metrics such as error magnitude or relative error are, unfortunately, sensitive to the input values.

To investigate the impact of the approximations on the quality of obtained results regardless of the values of the input items, we introduce another 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 $(2n+1)$ items is equal to the $(n+1)^{\text{th}}$ lowest value. The most important property of the optimized median algorithms is that the output always equals one of the input values. Let the output value equal to the j^{th} lowest value. To describe the quality of an approximate median function, we can introduce a new *error distance* defined as the distance of the item chosen as the output value (i.e. j^{th} lowest value) from the median (i.e. $(n+1)^{\text{th}}$ lowest value) calculated as $|j-n+1|$. Two additional metrics can be inferred from the error distance: *mean error distance* defined as the sum of error distances averaged over all input combinations producing an invalid output value and *worst case error distance* defined as the maximal error distance calculated over all input combinations. Note that it is not necessary to investigate all possible input combinations in practice. It is sufficient to calculate these metrics using the permutations of a set $S = \{-n, -n+1, \ldots, 0, \ldots, n-1, n\}$. In addition to that, the mean error distance can be calculated as the mean absolute error providing that we use the permutations of S. This simplification can be introduced because the median of S is zero and the distance between j^{th} lowest item (i.e. the value $j-(n+1)$) and $(n+1)^{\text{th}}$ lowest item (i.e. median value) is equal to $j-(n+1)$.

Table 3: Parameters of approximate 9-medians

Impl.	Oper. Reduct.	Error prob.	Distance error mean	worst
6-ops	84%	75.4 %	1.131 ± 0.847	3
10-ops	73%	63.5 %	0.778 ± 0.677	2
14-ops	63%	52.4 %	0.571 ± 0.583	2
18-ops	52%	34.9 %	0.361 ± 0.504	2
22-ops	42%	22.2 %	0.222 ± 0.416	1
26-ops	31%	11.1 %	0.111 ± 0.314	1
30-ops	21%	4.8 %	0.048 ± 0.213	1
34-ops	10%	5.6 %	0.056 ± 0.229	1
38-ops	0%	0.0 %	0.000 ± 0.000	0

Parameters of various implementations of approximate 9-median functions are shown in Table 3. After the identifier placed in the first column, the second column shows the improvement to the accurate median calculated as the relative reduction of the number of utilized operations. The third column contains the error probability. Mean distance error accompanied with the standard deviation is given in the fourth column. The last column contains the worst case distance error. The errors were calculated using all $9! = 362\,880$ permutations of $S = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$. The maximal possible worst case error is 4.

It can be seen that as the number of operations decreases, the error probability as well as distance error are increasing. Interestingly, the worst case distance error is not higher than 3. Because the equation to calculate the mean error distance does not include the correct output values, the mean error distance and the standard deviation suggest that the majority of the errors are caused by confusing the median value with such values of the sorted input sequence that are near to the median value. To illustrate this fact, we calculated histograms of error distribution. The histograms for three chosen implementations are depicted in Figure 1. If we reduce the number of operations by 21% (i.e. to 30 operations), the output value is determined correctly in 94.4%. In the rest of the cases (i.e. 5.6%), the output value is determined incorrectly as the 6^{th} lowest item of a sorted list of numbers. Because the median value corresponds with 5^{th} lowest item, the distance between median and output value is equal to 1. If the number of operations is reduced to 18, the worst case error increases to 2. According to the histogram, this error, which is caused by outputting 7^{th} lowest item, occurs in 1.19% of all input cases only. The remaining 33.7% erroneous outputs are caused by selecting 4^{th} or 6^{th} lowest item. Interestingly, distribution of the errors is asymmetric. This is more evident if we look at the distribution of errors for the implementation employing 6 operations.

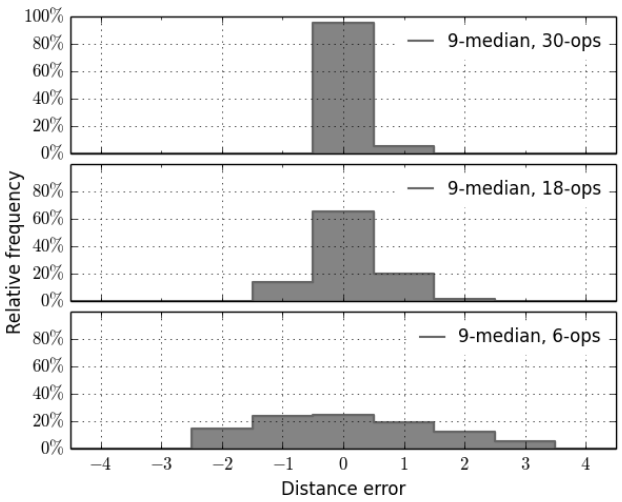

Figure 1: Error distribution for three different approximate 9-median implementations

The parameters of various approximate 25-median functions are shown in Table 4. As it is impractical to evaluate all possible input permutations for 25 inputs (there exist about $1.55 \cdot 10^{25}$ permutations), we randomly generated a subset of all the input permutations. We identified that it is necessary to generate at least 10^9 permutations to obtain results exhibiting error in the order of 10^{-3}. Hence, to calculate the errors, we used 10^{10} permutations of $S = \{-12, -11, \ldots, -1, 0, 1, \ldots, 11, 12\}$. The maximal worst case error is 12.

Again, there is a direct relation between the number of operations employed to the estimate median value and the error probability. Interestingly, the 22% reduction in the

Table 4: Parameters of approximate 25-medians

Impl.	Oper. Reduct.	Error prob.	Distance error mean	worst
10-ops	95%	91.1 %	3.321 ± 2.333	10
40-ops	81%	81.0 %	1.630 ± 1.281	7
70-ops	68%	67.9 %	0.986 ± 0.889	7
100-ops	54%	54.0 %	0.689 ± 0.753	6
130-ops	40%	26.6 %	0.299 ± 0.531	5
160-ops	27%	5.6 %	0.066 ± 0.292	5
170-ops	22%	2.8 %	0.032 ± 0.197	4
200-ops	9%	0.1 %	0.001 ± 0.037	2
220-ops	0%	0.0 %	0.000 ± 0.000	0

number of utilized operations leads to the lower number of invalid output values compared to the results obtained for 9-median. However, the worst case error increased to 4. The mean error distance indicates that the introduced errors have only a neglible impact on quality if the approximate median functions are utilized, for example, in image filtering. The mean error distance is less than 1 even if the number of operations is reduced by 68%. The histograms for three chosen implementations are shown in Figure 2.

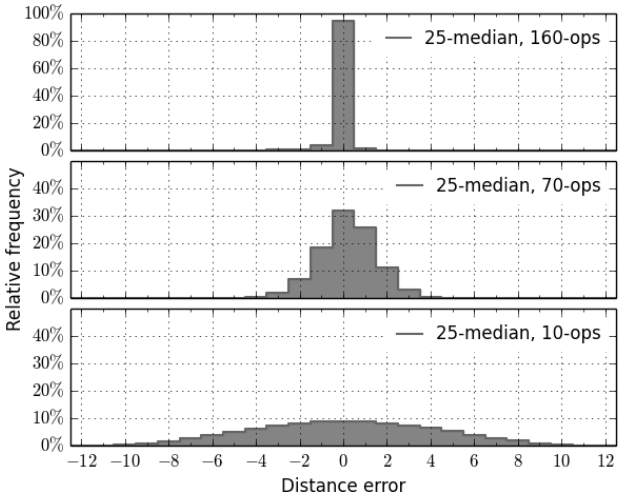

Figure 2: Error distribution for three different approximate 25-median implementations

4.4 Execution time and energy

The microcontrollers were programmed using the implementations discussed in the previous sections. Two non-functional parameters were measured: (a) the time that a microcontroller spends in a routine which computes the (approximate) median value, and (b) energy consumed by the microcontroller to execute this routine.

In order to perform these measurements, a specific program has to be implemented by the microcontrollers. It works as follows. Firstly, an input vector consisting of N words ($N = 9$ or $N = 25$) is randomly initialized and fed to the routines calculating the (approximate) median. Then, an infinite loop is executed, which contains calling of the routine calculating the (approximate) 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.

In order to correctly 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.

During the measurements, it turned out that energy consumption pattern remains almost invariable 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 2 mW for PIC16, 6.9 mW for PIC24 and 30.5 mW for ARM.

Table 5: Execution time and consumed energy of approximate 9-medians

Impl.	Time [μs]			Energy [nWs]		
	STM32	PIC24	PIC16	STM32	PIC24	PIC16
6-ops	2.8	54.5	170.5	86	377	342
10-ops	3.3	70.8	251.5	102	490	504
14-ops	3.9	86.8	336.5	118	600	674
18-ops	4.5	104.5	424.1	138	723	850
22-ops	5.0	116.7	487.8	151	808	978
26-ops	5.9	130.0	558.0	179	900	1118
30-ops	6.0	142.0	627.4	181	983	1257
34-ops	6.4	154.0	819.7	196	1066	1643
38-ops	6.9	165.5	885.0	210	1145	1774
qsort	28.5	1106.2	—	869	7655	—

Table 5 and Table 6 give the average execution time and energy consumption of various implementations of 9-median and 25-median functions. To demonstrate benefits of the median network, let us discuss the parameters of the accurate implementations at first. In the case of the accurate 9-median calculation at PIC24, the median network is 6.74 times faster than the quicksort algorithm and the consumed energy was reduced from 7655 nWs to 1145 nWs, i.e. by 85%. The median network is 4.13 times faster than quicksort on the STM32 and the energy was reduced by 76%. Similar results were obtained for the 25-median. The median network implemented on PIC24 is 4.06 times faster than the quicksort algorithm and the consumed energy was reduced by 75%. At STM32, quicksort algorithm exhibits 2.58 times worse execution time and by 158% higher energy consumption compared to the median calculated using 220 operations.

According to the results, there is nearly linear dependency between the number of operations used to estimate the median value and the execution time. As the execution time increases, the consumed energy also increases. Considering the parameters of the evolved median functions given in Table 3 and Table 4, we can easily determine that the proposed method has a great potential. For example, the 9-median implemented using 22 operations exhibits an error which is

Table 6: Execution time and consumed energy of approximate 25-medians

Impl.	Time [μs]		Energy [nWs]	
	STM32	PIC24	STM32	PIC24
10-ops	3.4	71.5	104	495
40-ops	8.1	188.5	246	1304
70-ops	13.3	303.0	406	2097
100-ops	17.3	401.6	528	2779
130-ops	22.1	491.2	673	3399
160-ops	27.4	581.4	836	4023
170-ops	29.1	609.8	888	4220
200-ops	34.8	698.3	1063	4832
220-ops	39.3	755.3	1200	5227
qsort	101.6	3067.5	3099	21227

negligible in the domain of signal processing, however, it enables to reduce the energy consumption by more than 25%. The 25-median implemented using 160 operations enables to reduce the energy by more than 23%. According to the distribution of errors shown in Figure 2, this implementation provides the output of high quality with very low percentage of erroneous outputs that are close to the median value.

Despite the fact that STM32 exhibits the largest current consumption in the run mode, it achieved the best results from the perspective of power savings. This is caused mainly by the fact that the STM32 incorporates a modern ARM-based RISC core having optimized instruction set and offering high computational power.

5. CONCLUSIONS

We presented a new approach to the approximation of software which is intended for microcontrollers. The method is based on CGP and exploits the fact that CGP can find a good trade off between the error and code size even if the code size is intentionally constrained. The method was evaluated in the task of 9-median and 25-median approximation intended for three different microcontrollers. Resulting approximations show a significant improvement in the execution time while the observed errors are moderate. Power consumption of approximate median functions linearly depend on the execution time. We also introduced a new method for error calculation which can be used in the future to evaluate approximate sorting networks and other routines.

Acknowledgment

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

6. REFERENCES

[1] A. Agapitos and S. M. Lucas. Evolving efficient recursive sorting algorithms. In *IEEE Congress on Evolutionary Computation*, pages 2677–2684, 2006.

[2] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. *Commun. ACM*, 58(1):105–115, Dec. 2014.

[3] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Proc. of the 18th IEEE European Test Symposium*, pages 1–6. IEEE, 2013.

[4] M. Harman and B. J. Jones. Search-based software engineering. *Information and Software Technology*, 43:833–839, 2001.

[5] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1–3):228–234, June 1990.

[6] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching (2nd ed.)*. Addison Wesley, 1998.

[7] J. R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3–4):251–284, 2010.

[8] W. B. Langdon and M. Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.

[9] K. Nepal, Y. Li, R. I. Bahar, and 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, pages 1–6. EDA Consortium, 2014.

[10] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `lulu.com`, available at http://www.gp-field-guide.org.uk, 2008.

[11] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174. ACM, 2011.

[12] L. Sekanina. Evolutionary design space exploration for median circuits. In *Applications of Evolutionary Computing*, LNCS 3005, pages 240–249. Springer, 2004.

[13] L. Sekanina and M. Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, 2005.

[14] L. Sekanina and Z. Vasicek. Approximate circuits by means of evolvable hardware. In *2013 IEEE International Conference on Evolvable Systems*, Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI), pages 21–28. IEEE CIS, 2013.

[15] Z. Vasicek and L. Sekanina. Evolutionary approach to approximate digital circuits design. *IEEE Trans. on Evolutionary Computation – In Press*, pages 1–13, 2015.

[16] D. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[17] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmailzadeh, and K. Bazargan. Axilog: Language support for approximate hardware design. In *Design, Automation and Test in Europe, DATE'15*, pages 1–6. EDA Consortium, 2015.