On Evolutionary Approximation of Sigmoid Function for HW/SW Embedded Systems

Milos Minarik(✉) and Lukas Sekanina

Faculty of Information Technology, IT4Innovations Centre of Excellence,
Brno University of Technology, Brno, Czech Republic
{iminarikm,sekanina}@fit.vutbr.cz

Abstract. Providing machine learning capabilities on low cost electronic devices is a challenging goal especially in the context of the Internet of Things paradigm. In order to deliver high performance machine intelligence on low power devices, suitable hardware accelerators have to be introduced. In this paper, we developed a method enabling to evolve a hardware implementation together with a corresponding software controller for key components of smart embedded systems. The proposed approach is based on a multi-objective design space exploration conducted by means of extended linear genetic programming. The approach was evaluated in the task of approximate sigmoid function design which is an important component of hardware implementations of neural networks. During these experiments, we automatically re-discovered some approximate sigmoid functions known from the literature. The method was implemented as an extension of an existing platform supporting concurrent evolution of hardware and software of embedded systems.

Keywords: Sigmoid · Linear genetic programming · HW/SW co-design

1 Introduction

There are many applications in which it is too expensive or impractical to employ a general purpose processor programmed to perform a given task. For example, in small electronic subsystems such as sensors, it is often impossible to perform basic signal processing on a processor because of its relatively high cost. On the other hand, a general-purpose processor could be acceptable in terms of cost, but it is insufficient in delivering expected computing power with a given power budget. This is clearly the case of complex machine learning algorithms such as deep neural networks (DNN) which are currently ported to low power electronic devices. Hence a boom of new hardware implementations of DNN is currently observed in which the requested performance is achieved by using multiple processing units and smart memory access optimized for energy efficiency [11].

As the target processing unit can show both combinational and sequential behavior, its implementation is based on (i) a data processing part composed of functional modules and registers, and (ii) a (micro)program stored in a memory.

J. McDermott et al. (Eds.): EuroGP 2017, LNCS 10196, pp. 343–358, 2017.
DOI: 10.1007/978-3-319-55696-3_22

The circuits of the data processing part can be configured, for example, in terms of the number of registers and their bit width, the number of modules, functions supported by each module, and interconnection options. The program then defines a sequence of operations over the preselected resources. The resulting HW/SW system can be programmed and configured to minimize power consumption, area or delay in a multi-objective optimization scenario. As this optimization task is difficult, a framework was developed which allows the designer to automatically evolve a control program together with the most suitable data processing circuits [10].

The objective of this paper is to extend the framework [10] in order to support the evolutionary design and optimization of elementary processing elements that are typical for recent neural networks implemented on a chip and demonstrate its performance in comparison with human-created designs. It is expected that the improved designs will lead to significant power reduction.

A clear disadvantage of current framework is the inability to effectively use the subsets of modules provided. If the framework were able to use arbitrary combination of modules, all these combinations would have to be specified in the instruction set. If there is a large number of modules, the instruction set can be quite extensive. This significantly increases the probability of disruptive mutation. Therefore it could be beneficial to introduce a mechanism enabling a better control of module utilization at the microinstruction level.

In order to optimize this kind of HW/SW systems, linear genetic programming (LGP) is used. The chromosome then contains two parts: (i) microinstructions to be executed and (ii) definition of the processing element (circuits of the data processing datapath). Resulting Pareto fronts then typically represent various design alternatives, where some solutions show better performance using more hardware resources and other solutions show better cost using fewer hardware components but more complicated program.

The proposed solution is evaluated in the task of sigmoid function approximation which is typically used as an activation function in the artificial neurons. Evolved approximate functions are compared with approximate sigmoid functions available in the literature. We show that a rich spectrum of sigmoid approximations can be obtained using the proposed approach.

The rest of the paper is organized as follows. Section 2 summarizes relevant state of the art and introduces the evolvable HW/SW platform. Section 3 is devoted to the extension of the framework enabling the deactivation of modules at microinstruction level. Experimental results are presented in Sect. 4. Conclusions are given in Sect. 5.

2 Previous Work

In this section similar approaches from the GP literature will be briefly reviewed. The proposed approach can be classified as a combination of genetic programming and evolvable hardware. Although we believe our approach is new and unique, there are some features similar to conventional evolutionary algorithms

based HW/SW co–design [4,5,14], co-evolution of programs and cellular MOVE processors [15]. The approach having the most features in common with the proposed solution is genetic parallel programming (GPP) [3]. GPP evolves efficient parallel programs by mapping a problem on parallel resources (ALUs), whereas the proposed method is more hardware oriented and allows optimizations at the level of the underlying digital circuits. Therefore we consider it more suitable for embedded systems where area, speed or power consumption is critical. The rest of this section contains a brief description of some basic terms (regarding the framework proposed in [10]) that will be used in the following sections.

2.1 Hardware

The HW part is composed of a configurable datapath which is controlled by a microprogram. The structure of the HW part can be seen in Fig. 1. Some parts are fixed and are not affected by the evolution process. These parts are drawn in gray. The other parts are subjects to the evolution. There is a set of registers connected to modules' inputs via multiplexers that are configurable by the microprogram. The outputs of those modules are then connected back to the registers using a set of decoders.

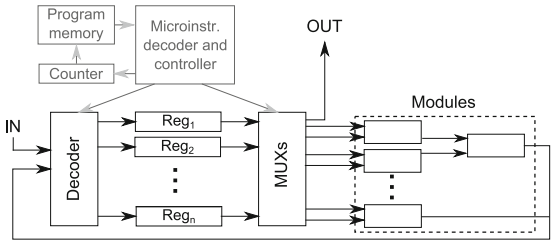

Fig. 1. HW architecture

Registers. The number of registers is given by the initial specification of the architecture and remains the same throughout the whole evolution. However the bit width of these registers can be affected by the evolution and it can range from 0 to the maximum width specified by the user. When the bit width of the register is set to 0, it is considered unused as it cannot influence the program execution. Therefore it is possible to let the evolution optimize the number of registers even if their number is constant.

Modules. The modules can be thought of as black boxes transforming the inputs to the outputs using an arbitrary function. Formally the module can be described as a 6–tuple $M = <n_i, n_o, a, p, d, f_t>$, where n_i is the number of module inputs, n_o the number of outputs, a is the area occupied by the module and p is its power consumption. Function d defines the processing delay of the module. Provided that **D** is the user chosen data type (integer or floating point

type), function d can be thought of as a projection $\mathbf{D}^{n_i} \rightarrow \mathbb{N}$ as it can use the values of inputs to asses the processing delay. This is useful in the case of modules realizing internally different functions based on the inputs provided. Finally function f_t is the output function transforming the inputs to the outputs. It can be described as $\mathbf{D}^{n_i} \times \mathbf{Q} \rightarrow \mathbf{D}^{n_o}$ where $\mathbf{Q}$ is the set of internal module states. The module can therefore retain some internal state during the program execution. However it is crucial that this state is reset between independent runs of the program.

Architecture. The HW part is described by following components:

$\mathbf{i}$	the number of inputs
$\mathbf{o}$	the number of outputs
$\mathbf{R} = \{r_1, r_2, \ldots r_r\}$	a set of registers
$\mathbf{w} : \mathbf{R} \rightarrow \mathbb{N}$	a function defining the widths of the registers
$\mathbf{A} = \{M_1, M_2, \ldots M_m\}$	a set of available modules
$\mathbf{u} : \mathbf{A} \rightarrow \{0, 1\}$	a function specifying module utilization

2.2 Software

Each program is composed of instructions $i_1, i_2, \ldots, i_s$, where s is the program size. Each of the instructions can consist of several microinstructions that get executed in the order defined by the instruction. The representation of the microinstruction is depicted in Fig. 2. The microinstruction is composed of the header specifying primarily the type of the instruction (e.g. branch instruction, reset instruction, or instruction utilizing the modules). The header also contains the information, which modules are used by the microinstruction. Right after the header there is a definition of a constant (that is used by some instructions) and definitions of input and output connections or values.

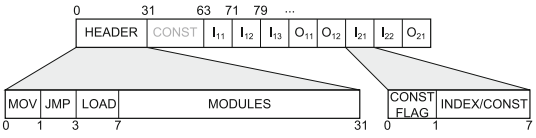

Fig. 2. Microinstruction format

2.3 Encoding and Search Method

The individuals are represented by chromosomes composed of integers. The first part of the chromosome describes the software part, where the program is encoded in LGP-like style [2]. The second part of the chromosome describes the hardware part. It contains the description of register bit widths, the usage of modules and the μ permutation encoding the order of modules. The μ permutation is the inversion sequence describing how many values precede the value at

particular position while being greater than that value. The main advantage of this encoding is its straightforward use with genetic operators, because it stays valid even after recombination or mutation. Therefore the software part stays valid even when the order of modules is changed, so there is no need to validate or fix the software part as would be the case if a direct encoding of modules order was used. The details can be found in [17].

The initial population is generated randomly by default. After the generation of initial population the evolution is started. It utilizes two–point crossover operator which performs crossover at the level of instructions in the software part and at the level of modules in the hardware part. A mutation operator implements several modifications of the chromosome. It can change the order of modules, their usage, bit width of the registers, instructions order and inputs, outputs and parameters of microinstructions. The selection is performed by a tournament method with the base of two.

The fitness of an individual is composed of four components: functionality fitness, speed fitness, area fitness and power consumption fitness. Due to the fact that there are multiple components of the fitness, the NSGA-II algorithm is utilized as it supports non-dominated sorting of candidate solutions and multi-objective optimization. However, due to the configurable design of the framework it is possible to change the algorithm easily or to select just a subset of predefined objectives.

3 Proposed Extension: Microinstruction-Level Modules Deactivation and a New Mutation Operator

Throughout previous experiments with the framework the disadvantage regarding the strategy in which modules are used was found. As already stated, the header of a microinstruction includes the information about the modules used. This information had to be hardcoded in the instruction set and could not be changed in any way by the EA. Therefore if the architecture should be able to perform various instructions utilizing different combinations of modules, all such instructions would have to be specified in the instruction set. For example, if the architecture employs 8 modules, there should be $\binom{8}{1} + \binom{8}{2} + \binom{8}{3} + \binom{8}{4} + \binom{8}{5} + \binom{8}{6} + \binom{8}{7} = 254$ instructions operating with the modules in the instruction set. The number of instructions can be easily handled as the instruction set is generated automatically. However, there are other problems imposed by the excessive number of instructions.

Let us analyze the following situation. The architecture contains 8 modules where two of them perform addition and another one performs multiplication. The framework discovered a candidate solution providing the expected outputs. This solution is depicted by black parts of Fig. 3. It is obvious that the output of the multiplier module is not used in the first instruction. Presuming the delay of the multiplier is longer than the delay of the adder, this solution is sub-optimal, as the first instruction takes longer than it has to. If the multiplier is disabled at the architecture level, it will be skipped during the execution. This will lead to

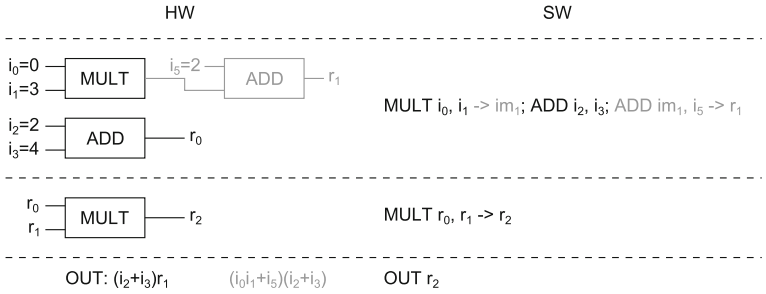

Fig. 3. Candidate solution. The instructions are separated by the dashed lines.

shorter delay of the first instruction and thus a better speed fitness. However, as the multiplier is disabled at the architecture level, it will be skipped also during the second instruction execution, therefore the r_2 register will not be set and the output of the third instruction will be wrong. The only possible way of achieving better speed fitness is the mutation that replaces the first instruction with the instruction performing only the addition. The mutation operator is implemented in such way it randomly chooses new instruction and generates random input connections for the modules. To achieve the desired effect, the mutation would need to choose the right one of 254 possible instructions and generate the same input connections (i_2, i_3) and output connection (r_0), what is quite unlikely.

To address this issue a modification of the SW part of the chromosome is proposed. This modification adds another property to microinstructions encoded in the SW part of the chromosome (the format of the instructions is not changed). It can be thought of as a bit string defining which modules are utilized by the microinstruction. This property is used during the microinstruction execution and if the module is not utilized by the microinstruction, it is skipped and its outputs are not available as the inputs for subsequent modules. Regarding this change, additional mutation operator was introduced that randomly flips the bits in aforementioned bit string of a particular microinstruction. The situation described in the previous paragraph can therefore be simply solved by deactivating the multiplier in the first microinstruction using this new mutation operator.

The downside of this approach is that the inputs of subsequent modules that were previously connected to the outputs of the deactivated module have to be connected to other points. Let's presume the candidate solution in Fig. 3 contains also the gray part. Presuming the i_0 input holds at zero value for all input samples, the multiplication in the first instruction is not needed. If the multiplier is disabled at the microinstruction level, the bottom input of the gray adder in the first instruction has to be connected somewhere else. If it gets connected to i_0 or to any of the uninitialized registers, the outputs will remain valid while the speed fitness increases. However, if it gets connected for example to i_1, the outputs will be wrong and the functional fitness will decrease.

On the other hand the proposed modification introduces some other advantages. During the experiments with the modified framework it was found that

the deactivation of a module at the microinstruction level does not only lead to better speed, area and power consumption fitness values, but also prevents the deactivated modules from spoiling the registers by their unneeded outputs. This side effect is important especially with respect to the architectures utilizing just a few registers. For example, if the output of the multiplier (in the first instruction in Fig. 3) was connected to r_0, it would overwrite the value stored by the adder and the outputs would not be correct.

Another advantage of the proposed extension is simpler generation of the instruction set. The simplest approach is to specify just the instruction utilizing all the modules and let the evolution disable the unneeded modules at the microinstruction level. There is also a possibility to divide modules to separate groups. For example, one instruction can utilize all the modules performing Boolean operations and another instruction can utilize the modules performing arithmetic operations as there is usually no point in combining Boolean and arithmetic modules in the scope of one instruction.

4 Experiments

The proposed framework will be evaluated in the task of sigmoid function approximation. In order to reduce a bias of the method, only the inputs and expected outputs will be provided in the training set.

4.1 Problem Description

The sigmoid function is defined as

$$y = \frac{1}{1 + \mathrm{e}^{-x}}$$

and has the derivative

$$\frac{dy}{dx} = y(1 - y)$$

The existence of the first derivative is crucial for artificial neural network training algorithms. Moreover, the sigmoid function is symmetrical with the point of symmetry at $(0, 0.5)$. Therefore, its value for negative values of x can be computed as $y(-x) = 1 - y(x)$.

A straightforward implementation of the sigmoid function is very resource demanding. Hence there is a need for its approximation. Most implementations of such approximations can be divided into three groups: piecewise linear approximations, piecewise second-order approximations and purely combinational approximations. There are also other approaches, e.g. lookup tables or recursive interpolation, but as stated in [16] they are outperformed by the three aforementioned approaches in terms of precision, area or speed, so they will not be further discussed in this paper.

The main goal of the experiments is to find out, whether the proposed framework is capable of finding some of these solutions on its own. As little information as possible was exposed to the framework so the solutions found can

be considered as new designs discovered by the evolution. Some decisions were made regarding the inputs and outputs representation. Although the framework is capable of working with floating point numbers, the HW implementation of floating-point arithmetics is more resource demanding compared with fixed-point arithmetics [12]. Hence fixed-point representation of inputs and outputs was chosen. In terms of bit width, we decided to use the representation with 6 fractional bits, as according to [16], this precision should be sufficient for implementing a reliable network forward operation.

4.2 Experiment 1: Using the Arithmetic Operations

The first experiment was based on the premise that the sigmoid function could be approximated on some interval by another function using less HW resources, but with required precision.

Experiment Setup. The modules allowed for use by the framework were 1 input module, 2 multipliers, 2 ALU modules and 2 bit shifters. The ALU modules can implement various basic arithmetical operations based on the function selection input. Namely these operations include addition, subtraction, incrementing and decrementing by one (in terms of the chosen fixed-point representation, where the increment is equal to 0.015625 for 6 bit inputs).

The training set was composed of 32 evenly distributed samples from the interval $<0; 4>$. The testing set was the whole set of possible inputs (i.e. 256 values). The parameters of the evolution are summarized in Table 1. The values of population size, crossover and mutation probabilities were chosen empirically based on several hundreds of runs with different values of these parameters. The maximum program length was chosen to be 10 instructions as during the aforementioned runs the output usually took place among the first ten instructions. The maximum logical time was inferred from the delays of the modules available and the maximum program length. The functionality fitness was defined as

$$f_o = \sum_{i=1}^{n_e} \frac{100}{n_s} \frac{1}{1 + (e_i - o_i)^2},$$

Table 1. EA parameters used for the first experiment

Parameter	Value
Population size	50
Max. generation count	200,000
Crossover probability	0.1
Mutation probability	0.7
Max. logical time	300
Max. program length	10

where e_i is i^{th} item from the sequence of expected outputs $(e_1, e_2, \ldots, e_n)$, o_i is the i^{th} output generated by the framework and n_s is the number of samples. The functionality fitness could therefore range from 0 to 100. Other parts of the fitness (speed, area and power consumption) were left to default (see [9]).

Results. 100 independent runs were carried out and the solutions found were then examined to asses their quality. The framework was able to find the solution in 19% of runs. The computational effort needed to find the solution was calculated according to [6]. To the best of our knowledge, no study has investigated the same problem. Therefore the sextic polynomial symbolic regression problem was chosen for comparison as it is a problem of comparable complexity and it has already been tested with one of the older versions of the proposed framework. Table 2 shows the computational efforts needed to find the solution by various approaches. Note there was no successful run in the sigmoid approximation experiment when conducted on the original framework [10], while it succeeded in the sextic polynomial regression experiment with the computational effort comparable to other methods. The sigmoid approximation could be therefore considered more complex problem than the sextic polynomial regression.

Table 2. Comparison of the computational effort with sextic polynomial symbolic regression

Problem	Method	Computational effort
Sextic polynomial	GPP $\mathcal{M}_{1,2}$ [7]	5,310,000
	GP [6]	1,440,000
	Original framework [9]	990,000
	GPP $\mathcal{M}_{8,8}$ [7]	540,000
Sigmoid approximation	Proposed framework	7,520,000
	Original framework	No solution

Afterwards the solutions found were examined. One of the solutions found is depicted in Fig. 4. It implements the formula

$$y = 1 - 2^{-1}(1 - 2^{-2}x)^2,$$

which is the expression realizing piecewise second-order approximation proposed by Zhang et al. [18]. Minor differences are present due to the fact the solution found by the framework implements the approximation only for the interval $<0; 4>$. Moreover, the approximation of Zhang et al. utilizes the nature of chosen binary encoding to replace the addition/subtraction by exclusive-or.

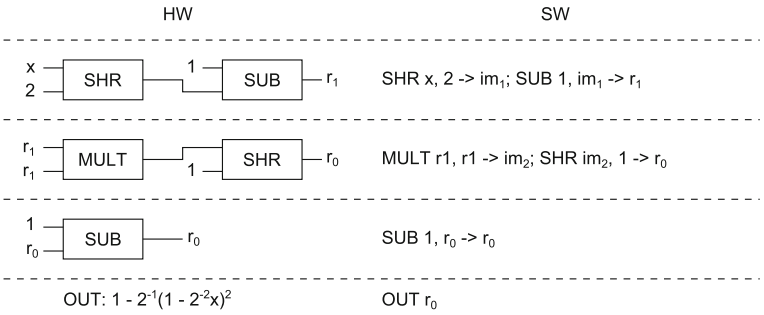

Fig. 4. Example of evolved solution in Experiment 1

Multiple variations of the correct solution were found. Most of them were computing the approximated value in the expanded form as

$$y = -2^{-5}x^2 + 2^{-2}x + 2^{-1}.$$

Some of those solutions were found to be sub-optimal in terms of area and speed, as they used multiplication by constant instead of a simple bit shift. Apart from that some solutions were found that even utilized both multipliers in parallel and therefore achieved lower area fitness but the highest speed fitness of all the solutions found. Figure 5 shows the non-dominated solutions discovered. The tradeoff between the speed and area fitness is clearly seen.

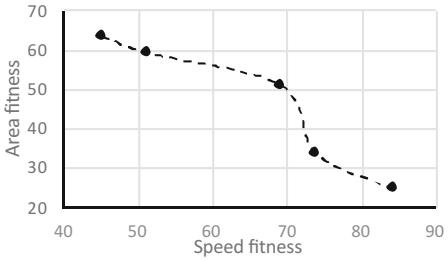

Fig. 5. Nondominated solutions for Experiment 1. The values were computed according to [9] and scaled from the $<0;1>$ interval to $<0;100>$ interval, where 0 is the worst fitness and 100 is the best (i.e. no modules used or zero execution time).

4.3 Experiment 2: No Multiplication

The framework was able to find a solution utilizing the multiplier module. However, multiplication is quite expensive in terms of the area used. Therefore, the next step was to find a solution that would not need the multiplier. One of such solutions (PLAN approximation) was proposed in [1]. This solution approximates the sigmoid by 4 linear segments (see Fig. 6).

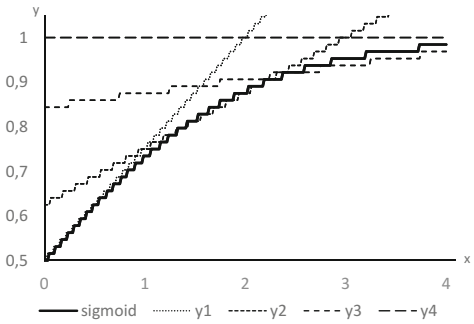

Fig. 6. PLAN approximation [1]

Each of the segments is used for some part of the interval. These segments can be described by the equations and appropriate intervals presented in Table 3. As can be seen, the multiplications by coefficients can be replaced by bit shifts. The HW implementation is, however, quite complex as it uses direct transformation of the inputs to outputs. Finding such a complex system at once would probably be nearly impossible. So the goal of our experiment was just to find a piecewise linear approximation of the sigmoid function.

Table 3. PLAN approximation of the sigmoid function [1]

Function	Interval
$y_1 = 0.25x + 0.5$	$0 \leq x < 1$
$y_2 = 0.125x + 0.625$	$1 \leq x < 2.375$
$y_3 = 0.03125x + 0.84375$	$2.375 \leq x < 5$
$y_4 = 1$	$5 \leq x$

Experiment Setup. The setup of the experiment remained almost the same as in previous experiment except the multipliers being removed. In the first experiment, only the first output for each sample was processed. However in the case of linear approximation, it could be beneficial to process more outputs and choose the one best approximating the sigmoid function for a given sample. This should enable the framework to evolve a program computing and outputting multiple linear approximations.

Results. 200 independent runs were performed and the results were examined. In 2.5% of runs a suitable solution was found. Outputs of one of the most precise solutions are depicted in Fig. 7. The solution differs from the original PLAN approximation. This is mainly due to the fact that the PLAN approximation is not bound only to interval $<0; 4>$ as the evolved solution is. The evolved solution utilizes this restriction to approximate the last segment of the interval

by constant 0.96875, whereas in PLAN approximation the constant segment is used for inputs $x \geq 5$. Moreover the gradient of the third segment differs from the PLAN approximation. That is, again, probably due to the interval restriction as the evolved solution does not have to approximate values between 4 and 5, where the gradient of the PLAN approximation is feasible.

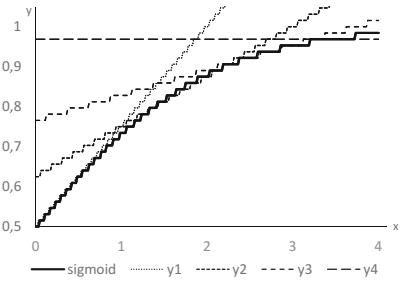

Fig. 7. Linear approximation A

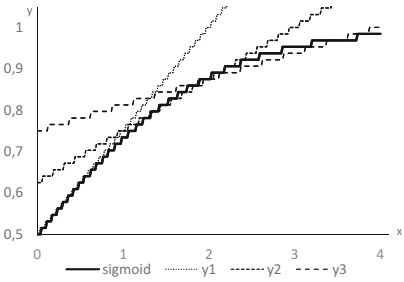

Fig. 8. Linear approximation B

Table 4. Description of segments – solution A

Function	Interval
$y_1 = 0.25x + 0.5$	$0 \leq x < 1$
$y_2 = 0.125x + 0.625$	$1 \leq x < 2.5$
$y_3 = 0.0625x + 0.765625$	$2.5 \leq x < 3$
$y_4 = 0.96875$	$3 \leq x$

Table 5. Description of segments – solution B

Function	Interval
$y_1 = 0.25x + 0.5$	$0 \leq x < 1$
$y_2 = 0.125x + 0.625$	$1 \leq x < 2.5$
$y_3 = 0.0625x + 0.75$	$2.5 \leq x$

Table 4 shows the description of the linear functions obtained by the analysis of SW part of the evolved solution. The intervals are given by the fitness function (i.e. the segment with the closest value is taken). In the case there were more possible points, the boundary of the interval was chosen to be the number with the least fractional bits for better readability of the table. Another solution is shown in Fig. 8 and the description of the linear segments is given in Table 5. It is similar to solution A, but uses just three linear segments. Moreover, it allows to design the HW implementation of such variation of a PLAN approximation, in which the offsets of the segments of solution B are more feasible.

Finally, the original PLAN approximation and the two proposed solutions were compared in terms of the average and maximum error. The results are listed in Table 6. The maximum error is the same but the average error is smaller for both evolved solutions. Therefore those solutions could be considered superior to original PLAN approximation in the interval $<0; 4>$. The framework therefore succeeded in evolving the solution suited specifically for this restricted interval and as [16] states, this interval is sufficient in many applications. The solution can therefore be considered as an improvement of an existing solution.

Table 6. Comparison of average E_{avg} and maximum E_{max} error

Approximation	E_{max}	E_{avg}
Plan approximation	3.13%	0.92%
Solution A	3.13%	0.61%
Solution B	3.13%	0.83%

4.4 Experiment 3: Combinational Approximation

The last approach is a purely combinational approximation. It is based on the fact that when both the input and output have a bit width restricted to only a few bits, it is possible to perform a direct bit-level mapping. More formally, the bit-level mapping can be expressed as a sum-of-products (SOP). The SOP can be minimized using a procedure such as Quine–McCluskey [8] and the result can be implemented by AND, OR and NOT gates.

Encoding. The bit widths should be as small as possible while still maintaining the required precision. As the previous experiments were carried out at the $<0; 4>$ interval, the inputs were chosen to have 2 integral bits, so the $<0; 4)$ interval is covered. The output is restricted to interval $<0.5; 1>$, so the outputs were chosen to have 0 integral and 6 fractional bits to provide the same precision as the previous experiments. The number of input fractional bits was decided to be 3 as, according to [16], it should provide a sufficient precision for neural network operation.

Experiment Setup. The input was chosen to have 2 integral and 3 fractional bits, therefore the modules allowed to use by the framework were 5 input modules (one for each bit) providing the bit value and its complement and 20 Boolean modules. Boolean modules can implement bitwise AND, OR, NAND or NOR. These modules were chosen to have 5 inputs. Four of them are used for actual inputs and the last one is used for Boolean operation selection. Four inputs were chosen as it is a typical number of binary inputs for a function that can be realized by a single look-up table in field programmable gate array (FPGA). The number of 20 Boolean modules is quite high compared to experiments performed with the previous version of the framework. It should, however, assure the evolution wouldn't be too limited by available resources and confirm that the proposed modules deactivation works as expected.

As only Boolean operations were used, the inputs were merged into 32 bit integers (according to [13]). The complete set of all the input combinations was chosen as the input set, as the goal of the experiment was to find a solution giving the correct outputs for all the input combinations. As there are only 5 input bits, the evaluation of whole training set could be done in one program execution. The outputs are processed in the same manner. The problem is that there are 6 outputs and it would not be possible to tell, which one should correspond to a particular expected output. Therefore the evolution was performed separately for individual outputs.

Modification of Boolean Modules Evaluation. After performing several runs, it was observed, that the candidate solutions tend to output 0 or -1 (all bits set). All possible input combinations are processed at once, which has an interesting side-effect. As it is known that none of the 32bit inputs nor the output is 0 or -1, these values can be ignored as they only spoil the computation. Thus the modification was made that the Boolean module ignores 0 values on its inputs when operating as AND/NAND and -1 values when operating as OR/NOR. This change reportedly lowered the computational effort by approx. two orders of magnitude.

Results. As the first fractional bit is known to be 1 for the whole positive domain, the first runs were performed for the second fractional bit of the output. The solution was found and compared to the solution sig_236 proposed in [16] (see Tables 7 and 8). The expressions in the tables use the notation from [16], where the input is of form $x_4x_3.x_2x_1x_0$. Redundant input bits in expressions in Table 7 are caused by redundant connections of corresponding module. This can happen as such connection influences neither the result correctness nor the speed or area fitness. Such connections could be easily identified and removed manually or some area would have to be assigned to the connections, so the framework would remove them while trying to minimize the area used. Another difference is the presence of $\overline{\overline{x_3} \vee \overline{x_0}}$ instead of $x_3 \wedge x_0$. According to the rules of Boolean algebra these expressions are equivalent, so it's not a difference in terms of functionality.

Table 7. Logic equations of the solution found for the second bit

Term	Expression
im_1	$NOR(\overline{x_3}, \overline{x_0})$
im_2	$AND(x_3, x_3, x_1)$
im_3	$AND(x_3, x_2, x_2)$
im_4	$OR(im_2, x_4)$
$Output$	$OR(im_4, im_3, im_1)$

Table 8. Logic equations for the second bit of sig_236p [16]

Term	Expression
p_2	$AND(x_4)$
p_4	$AND(x_4, \overline{x_3}, \overline{x_2}, x_0)$
p_{17}	$AND(x_3, x_0)$
p_{19}	$AND(x_3, x_1)$
p_{22}	$AND(x_3, x_2)$
$Output$	$OR(p_2, p_4, p_{17}, p_{19}, p_{22})$

The most important difference, however, is the absence of $x_4 \wedge \overline{x_3} \wedge \overline{x_2} \wedge x_0$. After the examination it was concluded that the absence of this expression is correct, because it gets minimized due to x_4 input as $x_4 \vee (x_4 \wedge \overline{x_3} \wedge \overline{x_2} \wedge x_0)$ minimizes to x_4, which is included. Presence of such expression in sig_236 could possibly be a mistake or some side-effect of the synthesis. This could happen e.g. in the case when some gates are shared by multiple outputs. In such case, it would be in accordance with aforementioned disadvantage of separate outputs evolution. However no evidence has been found to prove this. Afterwards the solutions were successfully found for all subsequent output bits, therefore the evolution succeeded in reinventing the sig_236p approximation presented in [16].

5 Conclusions

In this paper the framework for development of small application-specific digital embedded architectures was extended with the possibility to deactivate the modules at microinstruction level. The proposed extension was evaluated by evolving several distinct solutions of sigmoid function approximation. The extended framework was able to evolve variations of two sequential and one combinational well-known sigmoid function approximation algorithms. These results together with the results in [9] and [10] have shown that the framework can be used to find solutions for a wide variety of problems without the need of modifying the underlying algorithms. In most of the experiments, it was sufficient to specify the inputs and expected outputs, choose the modules available and define the functionality component of the fitness function.

The future research regarding this platform will deal with improving overall efficiency of the method. Another possibility would be to assess the extended framework on other complex problems, such as the evolution of convolutional kernels for DNNs.

Acknowledgments. This work was supported by the Czech science foundation project GA16-17538S.

References

1. Amin, H., Curtis, K.M., Hayes-Gill, B.R.: Piecewise linear approximation applied to nonlinear function of a neural network. IEE Proc. Circ. Dev. Syst. **144**(6), 313–317 (1997)
2. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, Berlin (2007)
3. Cheang, S.M., Leung, K.S., Lee, K.H.: Genetic parallel programming: design and implementation. Evol. Comput. **14**(2), 129–156 (2006)
4. Deniziak, S., Gorski, A.: Hardware/software co-synthesis of distributed embedded systems using genetic programming. In: Hornby, G.S., Sekanina, L., Haddow, P.C. (eds.) ICES 2008. LNCS, vol. 5216, pp. 83–93. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85857-7_8
5. Dick, R.P., Jha, N.K.: MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. IEEE Trans. CAD Integr. Circ. Syst. **17**(10), 920–935 (1998)
6. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
7. Leung, K.S., Lee, K.H., Cheang, S.M.: Parallel programs are more evolvable than sequential programs. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 107–118. Springer, Heidelberg (2003). doi:10.1007/3-540-36599-0_10
8. McCluskey, E.J.: Minimization of Boolean functions. Bell Syst. Tech. J. **35**(6), 1417–1444 (1956)
9. Minarik, M., Sekanina, L.: Concurrent evolution of hardware and software for application-specific microprogrammed systems. In: 2013 IEEE International Conference on Evolvable Systems (ICES), pp. 43–50 (2013). Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE Computational Intelligence Society

10. Minarik, M., Sekanina, L.: Exploring the search space of hardware/software embedded systems by means of GP. In: Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., García-Sánchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K. (eds.) EuroGP 2014. LNCS, vol. 8599, pp. 112–123. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44303-3_10
11. Misra, J., Saha, I.: Artificial neural networks in hardware: a survey of two decades of progress. Neurocomputing **74**(1–3), 239–255 (2010)
12. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Designs. Oxford University Press, Oxford (2000)
13. Poli, R., Langdon, W.B.: Sub-machine-code genetic programming. In: Advances in Genetic Programming, pp. 301–323. MIT Press, Cambridge (1999)
14. Shang, L., Dick, R.P., Jha, N.K.: SLOPES: hardware-software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable FPGAs. IEEE Trans. CAD Integr. Circ. Syst. **26**(3), 508–526 (2007)
15. Tempesti, G., Mudry, P.A., Zufferey, G.: Hardware/software coevolution of genome programs and cellular processors. In: First NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2006), pp. 129–136. IEEE Computer Society (2006)
16. Tommiska, M.T.: Efficient digital implementation of the sigmoid function for reprogrammable logic. IEE Proc. Comput. Digital Tech. **150**(6), 403–411 (2003)
17. Üçoluk, G.: Genetic algorithm solution of the TSP avoiding special crossover and mutation. In: Sixth Turkish AI and NN Symposium (TAINN VI), Ankara, pp. 57–62 (1997)
18. Zhang, M., Vassiliadis, S., Delgado-Frias, J.G.: Sigmoid generators for neural computing using piecewise approximations. IEEE Trans. Comput. **45**(9), 1045–1049 (1996)