Concurrent Evolution of Hardware and Software for Application-Specific Microprogrammed Systems

Milos Minarik and Lukas Sekanina
Brno University of Technology, Faculty of Information Technology
Brno, Czech Republic
Email: iminarik@fit.vutbr.cz, sekanina@fit.vutbr.cz

***Abstract*—Embedded systems often have to calculate some mathematical functions using iterative algorithms. When hard constraints are specified in terms of the area on the chip a possible solution is to implement the iterative algorithm by means of a microprogrammed digital circuit. In this paper, the first version of a new design framework is presented to automate the design and optimization of such microprogrammed systems. The framework utilizes evolutionary design and optimization techniques to find the most suitable implementation of the hardware architecture as well as the program for the programmable logic controller. The functionality of the proposed approach is evaluated using evolutionary design of three HW/SW systems under different constraints.**

I. Introduction

In many application domains (such as automotive or avionics industry), one can find small analog/digital measurement subsystems that are responsible for sampling signals from sensors, their basic digital processing and sending the results to a main controller. Their digital part, which is important for this paper, in fact implements very simple operations such as addition, multiplication, averaging or ultimately square root over the incoming samples. The subsystem is typically implemented as an application-specific integrated circuit (ASIC) and fabricated using a relatively obsolete technology (e.g. 180 nm) because there are many analog components and the digital part operates at low frequencies (e.g. 20 MHz). The relatively old technology also offers more reliable solutions because it ensures more stable physical properties of integrated circuits. These ASICs are produced in large volumes and have to be cheap.

In addition to meeting time requirements, there are strong constraints in terms of area occupied by the digital part. Because of that, it is impossible to implement the digital part as a general-purpose microprocessor. Even specialized iterative solutions based on the famous Cordic algorithm [1] are prohibited in some applications. The arithmetic functions are computed in iterations by means of a simple ALU and a set of registers. Their control is carried out by a programmable logic controller. The overall architecture is highly optimized for a particular application. The designer has to determine the number of registers and their bit width, the set of functions of ALU, interconnection options (allowed by multiplexers) etc. The hardware architecture influences the choice of the instruction set of the controller and vice versa. The program length and time of execution are determined by available hardware resources as well as the instruction set. Because of very specific and application dependent features, this kind of subsystems is predominantly designed and optimized manually which requires an extraordinary effort of a highly qualified designer.

A CAD tool supporting or even allowing the automatic design and optimization of such subsystems would be highly appreciated by practitioners. The tool could be classified as a HW/SW co-design tool in the scope of currently used CAD software. The goal of this paper is to present the first version of a new design framework that we have developed to automate the design and optimization of digital parts of the above-mentioned ASICs. The key idea behind the framework is to employ evolutionary design and optimization to find the most suitable implementation of the hardware architecture as well as the program for the programmable logic controller. In general, this is a very challenging task for evolutionary computing. However, because the target system is relatively small and its architecture can be predesigned to a wide extent, the evolutionary approach seems to be applicable.

We propose to employ *linear genetic programming* (LGP) to evolve programs for the controller [2]. The LGP chromosome is extended to encode selected features of the underlying hardware architecture. This allows us to evolve hardware and software together. Candidate solutions are then evaluated using the design framework that we have developed. As we present the first version of the framework, this paper concentrates on definition of a suitable design environment, search methods, and interaction of all components of the tool. The basic functionality of the proposed approach is evaluated using three case studies.

The rest of the paper is organized as follows. Section II surveys relevant work in the area of bioinspired hardware-software codesign. The proposed platform is introduced in Section III. Evolutionary aspects of the platform, such as problem encoding, genetic operations and fitness function, are described in Section IV. The platform is evaluated using three case studies which are presented together with obtained results in Section V. Finally, conclusions are given in Section VI.

II. Previous Relevant Work

The traditional design methodology would classify the proposed work under the umbrella of hardware/software co-design. Hardware/software co-design means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design [3], [4]. The hardware/software codesign methodology has to solve various problems (such as partitioning, scheduling, allocation), many of them known to be NP-complete. Hence various heuristics have been applied including evolutionary computation.

One of the most developed tools, the MOGAC system, employs a multioblective genetic algorithm that partitions and schedules embedded system specifications consisting of multiple periodic task graphs [5]. It optimizes price and power consumption while no limit is placed on the number of hardware or software processing elements in the architectures it synthesizes. Detailed analysis and comparison of partitioning algorithms, the most crucial part of the hardware/software codesing, has been done in many papers, e.g. [6], [7]. With the development of dynamically reconfigurable FPGAs, the hardware/software codesign methodology has been extended to support dynamically reconfigurable modules [8].

In the context of bio-inspired hardware, the most interesting relevant approach is Tempesti's hardware/software co-evolution of programs and cellular processors. Cellular processors are based on the so-called MOVE processors, where all computation is carried out by the functional units (adders, multipliers, register files, etc.) and the instructions simply move data to and from the functional units according the user program [9]. A genetic algorithm has been used to determine which parts of the program have to be implemented in hardware in order to satisfy some user-given constraints. The GA, in facts, solves the partitioning problem.

Another relevant approach, genetic parallel programming (GPP), has been developed to evolve parallel programs for processors containing multiple ALUs [10]. Based on LGP, it enables to automatically map a problem on parallel resources and evolve a corresponding parallel program. The approach is mainly focused on automated parallel programming. The hardware-oriented constraints that we have to deal with in our work are not taken into account.

A different GP-based approach to hardware/software codesign has been proposed by Deniziak and Gorski who evolved the co-design process itself using genetic programming, i.e. the chromosome represents the design decisions. The result of evolution is a method for constructing the target system [11].

Finally, in our previous work, we have developed a method to design iterative algorithms using Cartesian genetic programming [12]. However, no hardware related constraints have been considered.

III. Model of a HW/SW system

The proposed method is focused on HW/SW codesign of application specific microprogrammed architectures. The main goal is not to develop a framework that can be used to evolve a HW/SW system for arbitrarily complex problems. The framework is meant to design and optimize small microprogrammed systems for very specific problems with constraints on various attributes such as area, speed or power consumption. Typical usage of these HW/SW systems is e.g. capturing and preprocessing the data from a sensor. The model itself consists of three parts – microarchitecture, program and environment. All these parts will be briefly discussed in following subsections.

A. HW architecture

As can be seen in Figure 1, the architecture consists of several interconnected basic components. The whole HW/SW system operates iteratively by executing individual microinstructions. Therefore, the operation of this system can be described as:

1) Initialize the HW/SW system – all registers are set to their initial value (typically 0) and program counter is set to the first instruction of a program.
2) Fetch microinstruction from the program memory and increment the program counter (PC).
3) Decode the microinstruction and check if it is a branching instruction or ordinary instruction.
4) If it is a branching instruction, execute it, and modify the PC accordingly. Then go back to 2.
5) Set multiplexers and address decoder to provide interconnections as specified by the instruction.
6) If it is an I/O instruction, process the inputs and outputs as necessary and go back to 2. Otherwise, execute the modules specified by the instruction.
7) Update register values according to the outputs of modules.
8) Go back to 2.

Some of the parts (e.g. registers and modules) can be changed either by a user or by the optimization method (LGP, in our case), while the others are hard–coded and cannot be changed without modification of the framework's source code. Another noticable fact is that modules are not directly interconnected. Therefore, information can be moved among the modules just via registers. The advantage of such connection is the posibility to use the modules simultaneously. More detailed description of individual parts can be found in the following paragraphs.

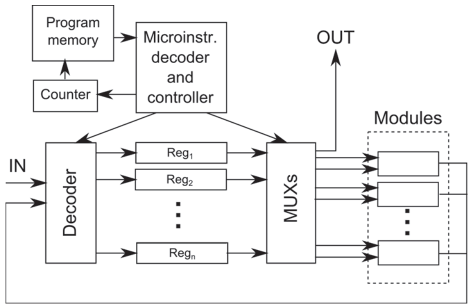

Fig. 1. HW architecture

Modules can be thought of as black boxes computing specific functions. More formally, a module is a 6–tuple $M =< n_i, n_o, a, p, d, t >$, where n_i is the number of inputs (typically 2 or 3), n_o is the number of outputs (typically 1), a is the area of the module, p is its power consumption, $d : \mathbb{D}^{n_i} \to \mathbb{D}$ is a function specifying processing delay and $f_t : \mathbb{D}^{n_i} \times Q \to \mathbb{D}^{n_o}$ is a function that sets module outputs according to the inputs provided and module's internal state $q \in Q$[1].

There are also other module parameters that don't influence the computation of its function, but can be used by

[1]$\mathbb{D}$ denotes a user-chosen data type.

the optimization framework. These parameters include the module area, power consumption and delay. The optimization framework uses these parameters to evaluate the fitness of the individual, so it has to be correctly specified for the evolution to succeed.

The architecture definition also contains the registers specification. This specification includes the register count and bit widths of all the registers. Various register widths are implemented by their masks. When the register widths are not to be concerned, the default mask can be used.

The last part of HW architecture that can be specified is the instruction set. The instruction set has to correspond with the modules used in the architecture. By default, all the instructions provided by the modules are used, so the possibilities of HW/SW system can be fully utilized. The system, however, does not have to utilize all the possibilities of all modules. It can, for example, use the adder module just to sum the content of two specific registers. In such case, there will be less interconnections between the module and the registers and the instruction decoder will be less complex as well. This approach is useful when some particular information about the algorithm to be found is known.

The whole HW part can be described by the following components:

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

B. SW architecture

The next part of the model is the SW architecture, which specifies the way the program is stored and executed. It is closely related to the instruction set described in previous subsection. However, this subsection mainly discusses the representation of the instructions, not their physical execution. The program consists of a sequence of instruction blocks $i_1, i_2, ...i_s$, where s is the program size. The instruction block serves as an envelope containing one or more microinstructions and corresponding parameters and inputs and outputs used by them. The instruction block can, therefore, be thought of as a single instruction composed of several microinstructions.

The microinstruction format is quite simple, so new instructions can easily be specified by user. As can be seen in Figure 2, there is a mandatory header, which specifies the type of the microinstruction and also the modules used. Then there might be a constant. Presence of a constant depends on microinstruction type. Then, the specifications of the inputs and outputs of all the modules follow.

Every input and output of each module used by the instruction is represented by one byte. This byte can specify the constant (only for module inputs), register index or a range specification used during the final microinstruction generation. If the input is a constant, the two highest bits are set to 10, therefore the constant has to fit into 6 bits. If the constant is greater than 63, it will have to be loaded via a register. If the

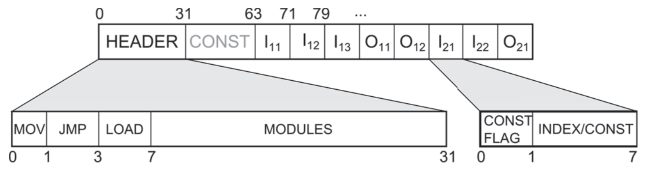

Fig. 2. Microinstruction format

input should be a register number, two leftmost bits have to be 00. In such case, the number directly represents the index of the register to be used. If two leftmost bits of the byte are set to 11 and the byte value is different from FF, it will be replaced by a randomly generated constant from the range from 0 to the value specified by the remaining 6 bits. The last option that can be used is random generation of a byte value. When the byte of the input or output is set to FF value, it will be replaced by random register index during the program generation.

As was stated above, the program itself is represented by one–dimensional array of instruction blocks. Execution of a program starts at its first instruction block. Then, the blocks are executed sequentially and the program counter is successively incremented unless a branching instruction (i.e. conditional or unconditional jump) is encountered. In such case, the program counter is modified to point to a given instruction block and the program execution continues from this point.

C. The model of the environment

The last part needed for successful simulation of the microprogram architecture function is its connection with the other parts of the resulting system. This can be accomplished by the environment part of the model. The environment can be thought of as a black box between the architecture outputs and inputs.

We currently support two ways of specifying the environment. The first is the usage of an XML file. This file contains the time series for individual inputs and also the timelines of the expected outputs. An example of the XML file specifying the environment can be found in Figure 3.

```
<environment>
  <inputs>
    <input index="0">
      <change time="237" value="85" />
      ...
    </input>
    ...
  </inputs>
  <outputs>
    <output index="0">
      <change time="330" value="34">
      ...
    </output>
    ...
  </outputs>
</environment>
```

Fig. 3. XML structure example

When a signal is set, it holds its value until another value is set. Therefore, the temporary values, which are present only for a limited amount of time, have to be explicitly specified by their start and end time.

The second supported approach is defining the environment by a reactive finite state machine which generates new inputs for the HW/SW system on the basis of the HW/SW system outputs and an internal state of the environment.

IV. Evolutionary framework

Having the model of the problem defined, the evolution framework can be specified. This framework is supposed to serve as a tool for evolutionary design and optimization of HW/SW systems. First of all, it is crucial to define which parts of the HW/SW system can be changed by the evolution framework. Considering the HW, there are three parts that should be included – registers, modules and instruction set. The interconnections between the registers and modules do not have to be included in the process of evolution, because they can easily be derived from the modules, registers and microinstructions used. In terms of SW there is only one thing to be evolved – the program itself.

The next step is determining the search method that should be used. Considering the fact that the SW part of the system is represented by one–dimensional array of instruction blocks and the programs are executed in sequential manner, Linear Genetic Programming (LGP) [2] seems to be the best solution. However, to be able to evolve also the HW part of the system, LGP has to be modified. The modifications will be discussed in following sections.

A. *Individual encoding*

Since the individual has to encode both HW and SW part, the chromosome structure has to be heterogenous.

The HW part of the architecture represents the usage and bit widths of the registers and the usage of the modules. The encoding needs to fulfill some special requirements. The most important requirement is that the program stays valid independently of the HW architecture changes. For example, when a register is removed, the program still has to be valid and executable. The proposed method deals with this problem in a quite straighforward way. Each register has its width specified. When the width is set to zero, the effect is the same as if the register was removed, but all the instructions can be executed the same way as before. If the module is to be removed, it is just marked as inactive. During the instruction execution the outputs of inactive modules are ommited. Using this approach, the HW architecture can be represented as a heterogenous array containing all the forementioned information.

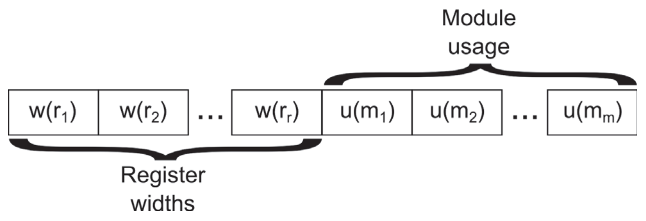

Fig. 4. HW encoding

Figure 4 shows that the HW part of a chromosome contains r integers representing register widths and m bit values representing the utilization of modules. These individual numbers are considered as genes, in terms of LGP, and their counts are always constant. This property is particularly useful when performing the crossover operation. The SW part of the chromosome is encoded in such a way that individual instruction blocks are considered as genes and the whole program sequence equals to the SW part of a chromosome.

B. *Generating the initial population*

Considering the chosen individual encoding scheme, the program can be generated very simply. It is sufficient to randomly generate the instructions according to the maximal program length specified. Some of the instructions use a parameter, so it also has to be specified. Since the parameter depends on instruction type, the valid parameter range has to be specified for each instruction type. These ranges have some default values, but all of them can be redefined by user. The specification of parameter ranges can be very difficult since the program will be modified during the evolution and the instructions can change their positions. For example, when a jump instruction is generated with jump offset of 10 instructions and during the evolution this instruction is moved to be e.g. just 4 instructions from end of the program, the instruction will be invalid, because it is pointing outside the program. This issue can be addressed by checking program validity during the evolution. As this operation can be very time consuming, another approach was chosen. When a jump exceeding the program range is detected, it is limited to the beginning or end of the program depending on its direction. Therefore, the range can be the same for all the jump instructions regardless of their position in a program. Given this condition the parameters can be generated using Gaussian distribution having the mean value in a center of the range and standard deviation determined by the 3σ rule.

C. *Computing the fitness function*

As the problem is inherently multiobjective, the fitness is not just a single value. There are four objectives to be reflected in the fitness – speed, area, power consumption and functionality (i.e. correctness of the output signals). The overall fitness can be represented as:

$$\vec{f} = (f_a, f_p, f_s, f_o)$$

The area is expressend in terms of the fitness value as

$$f_a = \frac{1}{1 + \sum_{i=1}^{m} \begin{cases} a_i & \text{if } u(M_i) = 1 \\ 0 & \text{if } u(M_i) = 0 \end{cases}} = \frac{1}{1 + \sum_{i=1}^{m} u(M_i) a_i},$$

where a_i is the area of corresponding module M_i. The function takes into account, via the u predicate, only the modules used in the final phenotype.

The power consumption fitness can be evaluated in a similar manner as

$$f_p = \frac{1}{1 + \sum_{i=1}^{m} u(M_i) p_i},$$

where p_i is power consumption of respective module M_i.

The speed of computation depends just on the processing time:

$$f_s = \frac{1}{1+T},$$

where T is total processing time of the program.

The functionality depends on the HW/SW system outputs generated during the execution of a program.

$$f_o = \sum_{i=1}^{n_e} \begin{cases} 1 & \text{if } e_i = o_i \\ \frac{1}{1+(e_i-o_i)^2} & \text{otherwise} \end{cases} = \sum_{i=1}^{n_e} \frac{1}{1+(e_i-o_i)^2},$$

where e_i is i^{th} item from the sequence of expected outputs $(e_1, e_2, ...e_n)$ and o_i is the i^{th} output generated by the HW/SW system.

The above-mentioned functions are predefined in the design framework. However, they can be modified by user. It can be useful e.g. when the user is concerned only in satisfying some boundary conditions. For example the fitness dealing with the time of execution can be specified as

$$f_s = \begin{cases} 1 & \text{if } T < T_{max} \\ \frac{1}{1+T} & \text{otherwise} \end{cases}.$$

This means that the individuals with processing time less than the maximal allowed time T_{max} will have the highest fitness while other individuals will have the fitness proportional to their execution time in the range from 0 to 1.

The f_o fitness is usually obtained by comparing the output values produced by an individual with expected output values. This comparison can be done in several ways ranging from simple Euclidean distance up to complex functions considering also the relations between individual outputs. The choice of the fitness function can strongly influence the success rate and speed of LGP.

Because the fitness is represented by a vector of components, there has to be an algorithm to compare two fitness values. There are many possibilities of doing such comparison. One of them is choosing the importance of components (i.e. sorting them). Then two vectors can be compared easily by comparing their components in a chosen order. The drawback of this method is the need to sort the components by their importance, because the evolution will always try to prefer search in one direction. However, there is often a need to find solutions from different parts of the search space, e.g. fast but large solutions and small but slow solutions or their combinations. The designer can then choose the solution which suites best some particular use. In short, all the nondominated solutions should have the highest fitness.

There are multiobjective algorithms that address this issue, e.g. NSGA–II (nondominated sorting genetic algorithm – see [13]) or ISMAUT (Imprecisely Specified Multi–Attribute Utility Theory – see [14]). The proposed method uses NSGA–II as the method for finding nondominated solutions. It sorts the individuals into ranks by putting the nondominated solutions into the first rank, removing them from the set, then taking nondominated solutions from the remaining set and placing them in the second rank etc. Solutions inside a rank are then sorted by mean Euclidean distance from the other solutions in the same rank, where the solutions with bigger mean distance are considered better. During the expriments with the proposed method, one disadvantage had been observed. At the beginning of the evolution when none of the individuals has the functionality f_o greater than zero, the solutions occupying less HW resources are preferred. This consequently leads to a state when all the individuals have minimal HW and the solution cannot be found due to insufficient HW resources. To address this issue, only functionality can be taken into account at the beginning of evolution. Once it reaches a predefined value, NSGA–II is employed.

D. Selection

Having the fitness function and comparison method defined, the selection can be performed. There are several selection methods, that can be used. Due to the use of NSGA–II algorithm the fitness proportional selection cannot be directly used, because the fitness vector cannot be converted to scalar value in a simple manner. Tournament selection was chosen from the remaining methods because of its advantages, namely the possibility to easily change the selection pressure.

E. Crossover

After the individuals selection, the crossover takes place. Considering the forementioned chromosome structure there are several crossover methods that can be used. The proposed framework allows specifying the number of parents and the number of crossover points. The two point crossover was chosen as default, as it led to the best results on most of the experiments performed so far.

F. Mutation

Because the chromosome of the individual is heterogenous in the proposed encoding, certain mutation types can be performed only on some genes (e.g. a HW mutation can be executed only on genes from the HW part of a chromosome).

HW mutation can basically influence registers, modules and instruction set. The first of the mutation types modifies the width of the register. The register is randomly chosen using the uniform probability distribution, so the probability of modification is the same for all the registers. Then, its width is changed by n_{rand} bits, where n_{rand} is generated using the Gaussian probability. The Gaussian mutation was chosen, as it is considered superior to the traditional bit-flip mutation in most cases regarding the numeric value mutation [15]. The resulting width must lie in range from 0 to maximal allowed width. When the register has the width of 0 bits, it is considered unused. Hence, this mutation can effectively reduce the number of registers.

The second type modifies the module usage. As availability of each module is represented by one bit, the module usage can be changed by simply flipping its respective bit. The bit that will be flipped is randomly chosen using the uniform distribution.

The last part of the architecture that can be affected by mutation is the instruction set. There is a possibility to change the instruction set, e.g. by adding new instruction that will contain a sequence of two already existing instructions.

Such instruction can become a building block that speeds up program evolution. To be able to generate suitable instructions some statistics of used instructions and their relative order in individuals with higher fitness would need to be done. However, this functionality has not been implemented yet.

SW mutation can modify the program in terms of instructions, their types, parameters, inputs/outputs specification and order. First of all, the instruction block is selected using the uniform distribution. Then, the specific microinstruction can be selected within the block again using the uniform distribution. All the microinstructions, therefore, have the same probability to be selected. When the microinstruction is selected, the mutation types that will be applied are chosen by generating random numbers and comparing them to probabilities of mutation types.

The first mutation type changes the microinstruction as a whole. It selects the microinstruction from the instruction set and checks the valid range of the parameters and inputs/outputs indices against the parameters of the original microinstruction. If the parameter is out of range, the new one is generated the same way as during the initial population generation.

The second type of mutation changes only the parameter of the microinstruction. The original parameter is changed by the value randomly generated using Gaussian distribution while preserving the validity of the parameter value.

The third type changes the input and output indices of the microinstruction. New index is generated in compliance with constraints determined by the instruction specification contained in the instruction set.

The last type of mutation doesn't modify the microinstruction itself, but its position in a program. It randomly selects the position offset using Gaussian distribution and then moves the whole instruction block by the offset generated.

Operator probability adaptation: The last important thing about genetic operators is the specification of their probabilities. Choosing the most suitable probabilities of crossover and mutation can be quite complex task by itself. Proposed method addresses this problem by introducing self–adaptive probabilities. At the beginning of the evolution the operator probabilities are set to initial values and the evolution starts. Every time some operator is carried out, the fitness of the modified individual is computed and compared with the fitness of the original individual. If it is greater, the score of respective operator will be increased by one. After the specified number of generations the usefulness of individual types is evaluated. The most useful types (i.e. those with the highest score) will get their probabilities increased, whereas the probabilities of unusefull types will lower.

V. Experimental results

Several experiments were carried out to verify the proposed method. This section contains some of them and also a comparison with similar methods. It is, however, important to keep in mind that the proposed method has to also deal with evolution of hardware part whereas available methods are used to develop just the software part. Another reason for performing the experiments was also the need to find out some useful information about the method itself, e.g. what are the most useful operators, whether a population size influences speed of the evolution etc. The experiments chosen are quite simple, mainly because the proposed method is still under development and these experiments were supposed to verify core functionality of the method. More complex experiments will be performed in the future. The second reason for choosing these experiments was the need to compare the proposed method with similar methods. The experiments were, therefore, selected from the limited set of experiments common to the methods compared.

Preparation of an experiment is quite straightforward. First of all, the environment, available modules, registers count, fitness function and termination condition must be specified. Available modules and registers count can be overestimated, because the evolution will optimize the architecture. Remaining parameters of LGP can be left at their default values.

Common experiment parameters If not stated otherwise the experiments were performed with the parameters listed in Table I. Parameters that differ among the experiments are specified in each experiment's subsection. Two types of modules were used in experiments. The first one (entitled ALU) implements a simple ALU that can perform addition, subtraction, incrementation and decrementation. The other one (entitled MD) is a module implementing multiplication and division operations.

TABLE I. Common experiment parameters

Parameter	Value
Register count	3
Default register width	32b
HW/SW system inputs count	1
HW/SW system outputs count	1
Program size	20 instruction blocks
Population size	5
Maximal number of generations	200,000
Crossover probability	0
Mutation probability	0.7

A. *Fibonacci series*

The goal of this experiment was to find a microprogrammed architecture generating the first 11 numbers of Fibonacci series. This count was chosen to illustrate the possibilities of hardware optimization. In this case the HW/SW system has no inputs and one output. Two instances of ALU module were chosen as the only available modules. The functionality fitness function was defined as

$$f_o = \sum_{i=1}^{n_e} \begin{cases} 1 & \text{if } e_i = o_i \\ 0 & \text{otherwise} \end{cases} .$$

After performing several runs, some solutions were found. Software part of one of these solutions is shown in Figure 5 (operations with no effect have been omitted to make the figure clear). This solution uses only one of the two available modules. The registers widths are 6 and 7 bits. Notice that the registers have the smallest possible widths to be able to store the last two numbers to be found. Therefore, it is the optimal solution in terms of total area used.

Then the evolution was set to stop when the fitness value of the best individual reaches 11 and the total area of the

```
LOAD r1, 1
ADD r1, r2 -> r2
OUT r1
OUT r1
ADD r1, r2 -> r1
OUT r1
ADD r1, r2 -> r2
OUT r2
JMP -4
```

Fig. 5. Software part of one of the solutions of Fibonacci experiment

microarchitecture is minimal. After defining these parameters several runs were performed with various combinations of crossover and mutation probabilities. Figure 6 shows the computational effort (calculated according to [16]) obtained for each combination of mutation and crossover probability using 20 independent runs. It can be seen that the computational effort increases with increasing crossover probability. Hence, the crossover should not be used in this case. The probability of mutation should be approximately 0.7.

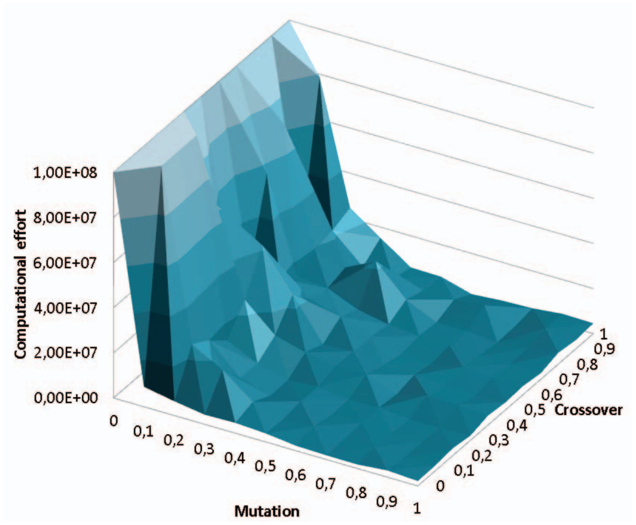

Fig. 6. Computational effort for Fibonacci experiment with respect to various probabilities of crossover and mutation

Next modification of the experiment was supposed to show how the population size affects the computational effort. 50 independent runs were performed with various population sizes, while the number of evaluations remained constant. For each population size some runs were constrained by a total area allowed on a chip. This constraint was implemented by a termination condition, which was set to stop the evolution when all the ouputs were correct and the area was minimal. Some other runs did not take this constraint into account. The rest of the LGP parameters remained identical with the previous experiment. Figure 7 shows that the computational effort decreases with decreasing population size regardless of whether the area constraints are taken into account or not.

B. Squares

This experiment was supposed to find a microarchitecture outputting the squares of the input values. The environment

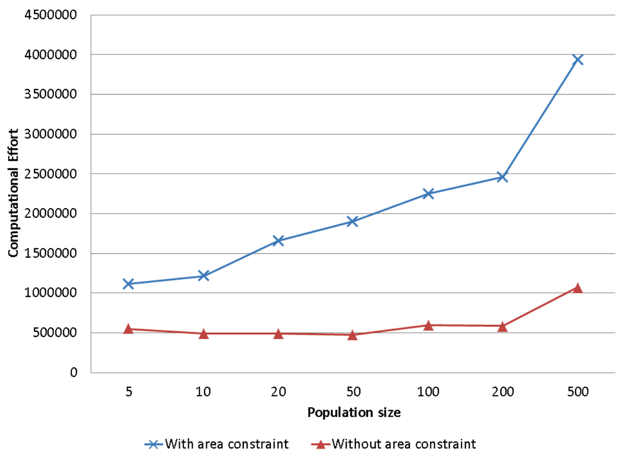

Fig. 7. Comparison of computational effort for Fibonacci experiment regarding the different sizes of population and HW optimization

was specified in such way that the input value stays unmodified until a new value is available at the circuit output. Then, the input value is changed and stays at this value until next output is provided. The fitness function is defined in the same way as in the previous experiment. The termination condition was set to stop the evolution when the best individual reaches the maximal possible fitness in terms of correctness. The experiment was split into two parts. In the first part the allowed modules were 1xMD and 1xALU. As can be seen in Table II, the evolution was able to find a solution in both cases, although the computational effort is significantly higher in the second case due to the fact it has to evolve a program consisting of two nested loops – the inner one computing the square and the outer one iterating through the inputs.

TABLE II. COMPARISON OF COMPUTATIONAL EFFORT OF SQUARES EXPERIMENT WITH RESPECT TO AVAILABLE MODULES

	Computational Effort	Ratio
With MD	32,000	1.0
Without MD	17,790,000	555.9

C. Sextic polynomial

This experiment was chosen mainly to compare the proposed method with existing methods in the field of symbolic regression. The environment was defined the same way as in the previous experiment. The polynomial to be found was $x^6 - 2x^4 + x^2$. The available modules were chosen so all the operations typically used by other methods were implemented (i.e. 1xADD module and 1xMD module). The termination condition was set to stop LGP when all the values of the training set consisting of 20 samples were found by a candidate program. The computational effort estimated from 100 runs is shown in Table III together with computational efforts of other methods for comparison.

VI. CONCLUSION

We have shown that the proposed method can evolve microprogrammed architectures capable of solving some of typical problems that were approached by GP in the past.

TABLE III. Comparison of experimental results with other methods

Method	Computational Effort	Ratio
GPP $\mathcal{M}_{1,2}$ [17]	5,310,000	5.4
GP [16]	1,440,000	1.5
Proposed solution	990,000	1.0
GPP $\mathcal{M}_{8,8}$ [17]	540,000	0.5

In some cases the proposed method provides better results in terms of computational effort. The method, therefore, seems promising for further exploration.

The main goal of our upcoming research will be improving the method to be usable in real-world problems and under hard constraints such as iterative computation of $\sqrt{x^2 + y^2}$ using only addition, subtraction and shift operations. We will try to achieve this goal by implementing some techniques that proved useful in decreasing the computational effort needed to find a solution, such as automatically defined functions [16]. Another subject to explore could be parallelization in terms of speeding up the evaluation but also lowering the computational effort (see [17]).

Acknowledgment

This work was supported by the Czech science foundation project P103/10/1517.

References

[1] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330 –334, 1959.

[2] M. Brameier and W. Banzhaf, *Linear Genetic Programming*. Berlin: Springer Verlag, 2007.

[3] G. D. Micheli and R. K. Gupta, "Hardware/software co-design," *IEEE MICRO*, vol. 85, no. 3, pp. 349–365, 1997.

[4] M. Chiodo, P. Giusto, A. Jurecska, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, no. 4, pp. 26–36, 1994.

[5] R. P. Dick and N. K. Jha, "Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 920–935, 1998.

[6] M. López-Vallejo and J. C. López, "On the hardware-software partitioning problem: System modeling and partitioning techniques," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 3, pp. 269–297, 2003.

[7] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partitioning in hardware-software codesign," *Design Autom. for Emb. Sys.*, vol. 6, no. 4, pp. 425–449, 2002.

[8] L. Shang, R. P. Dick, and N. K. Jha, "Slopes: Hardware-software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable fpgas," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 3, pp. 508–526, 2007.

[9] G. Tempesti, P.-A. Mudry, and G. Zufferey, "Hardware/software coevolution of genome programs and cellular processors," in *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2006)*. IEEE Computer Society, 2006, pp. 129–136.

[10] S. M. Cheang, K. S. Leung, and K. H. Lee, "Genetic parallel programming: design and implementation," *Evol. Comput.*, vol. 14, no. 2, pp. 129–156, 2006.

[11] S. Deniziak and A. Gorski, "Hardware/software co-synthesis of distributed embedded systems using genetic programming," in *Evolvable Systems: From Biology to Hardware*, ser. LNCS, vol. 5216. Springer, 2008, pp. 83–93.

[12] M. Minarik and L. Sekanina, "Evolution of iterative formulas using cartesian genetic programming," in *Knowledge-Based and Intelligent Information and Engineering Systems - 15th International Conference, KES 2011, Part I*, ser. LNCS, vol. 6881. Springer, 2011, pp. 11–20.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[14] B. S. Ahn, "Multiattribute decision aid with extended ismaut," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 36, no. 3, pp. 507 –520, may 2006.

[15] R. Hinterding, "Gaussian mutation and self-adaption for numeric genetic algorithms," in *Evolutionary Computation, 1995., IEEE International Conference on*, vol. 1, 29 1995-dec. 1 1995, pp. 384–389.

[16] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, 1994.

[17] K. S. Leung, K. H. Lee, and S. M. Cheang, "Parallel programs are more evolvable than sequential programs," in *Proceedings of the 6th European conference on Genetic programming*, ser. EuroGP'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 107–118.