
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER SYSTEMS**
**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**


# CONCURRENT EVOLUTIONARY DESIGN OF HARDWARE AND SOFTWARE

**SOUBĚŽNÝ EVOLUČNÍ NÁVRH HARDWARU A SOFTWARU**

**PHD THESIS**
**DISERTAČNÍ PRÁCE**


**AUTHOR** **Ing. MILOŠ MINAŘÍK**
**AUTOR PRÁCE**

**SUPERVISOR** **Prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
**ŠKOLITEL**



**BRNO 2017**

# Contents

# Chapter 1

# Introduction

Arthur Samuel, machine learning pioneer, stated in 1983 that the main goal of machine learning and artificial intelligence is "to get machines to exhibit behaviour, which if done by humans, would be assumed to involve the use of intelligence" [39]. Genetic programming (GP) is one of the methods developed as an attempt to fulfil this vison. GP was designed to automatically generate desired programs without asking the user to specify how to do it. GP has been used to solve a wide range of practical problems and produce a number of human-competitive results in different fields. GP is predominantly used to create computer programs, but for example, competitive digital circuits were evolved as well [38].

An interesting and practically untouched question (see Chapter 2.4.1) is whether for a given problem, GP can generate a highly optimized programmable computational model (platform) together with a program running on the platform, solving the problem and satisfying all constrains such as on the area and speed. In a multi-objective scenario, the user would obtain a set of non-dominated solutions showing various trade-offs between resources (the area, power consumption) and performance (the speed of execution). From the computer architecture perspective, this problem can be seen as a concurrent development of hardware and software, simply, HW/SW codesign. This thesis explores the ways how to evolve hardware platforms together with programs in the case that the specification is given in terms of a set of desired input-output responses.

This research was motivated by practical problems in the area of design and optimization of small industrial HW/SW systems. During the last three decades the use of electronics in devices of everyday use started to grow rapidly. These devices are often constructed as embedded or cyber-physical systems based on universal processors. It is predicted that the number of such systems will be significantly growing with the development of the internet of things (IoT) [55]. However, 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. Even specialized iterative solutions based on the famous Cordic algorithm [51] are prohibited in some applications due to hard area constraints.

Due to the limitations mentioned, microprogram architectures are often employed in such 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.

In general, the design and optimization of such HW/SW systems is a very challenging task for evolutionary computing. However, because we will deal with relatively small-size applications and some parts of the hardware architecture can be predesigned, the evolutionary approach seems to be, in principle, applicable.

## 1.1 Research Objectives

The main objective of this thesis is to investigate whether and to what extent an evolutionary design based method can evolve a program concurrently with a programmable computational platform in such a way that, at the end, the user is provided with various optimized HW/SW implementations showing competitive trade-offs between key system parameters. As this ultimate objective is a really challenging research problem, it was split to following partial sub-goals:

1. Identify a suitable computational model and evolutionary program design method that can be combined to automatically evolve and optimize HW/SW systems from behavioral specifications.

2. Develop and implement an experimental framework that enables automated design, optimization and evaluation of HW/SW systems.

3. Perform initial experiments with the framework using small problem instances and compare the results with outcomes of available evolutionary computation based methods.

4. Extend and tune the framework in order to (i) eliminate the problems encountered when performing the initial experiments and (ii) solve more complex problem instances.

5. Evaluate the extended framework in the design and optimization of more complex HW/SW systems.

# Chapter 2

# Definitions

The thesis deals with the evolutionary design and optimization of microprogram architectures. As this topic is closely related to hardware and software evolution, this chapter contains the basic information about the needed parts. Chapters 2.1 and 2.2 provide an introduction to evolutionary computation and its use in optimization of real world problems. There is an overview of some common variants of evolutionary algorithms. Then the symbolic regression problem is described and genetic programming (and its common variants) is introduced. Finally, the importance of multicriterial optimization is discussed, together with an NSGA II algorithm that can be used for this purpose. The hardware related part is described in Chapter 2.3. After introducing basic architectures available for hardware design, their pros and cons are discussed. Then the design process of a system containing HW and SW is described and the problems connected to this field are discussed. Finally, Chapters 2.4 and 2.4.1 summarize the state of the art in the use of evolutionary techniques in the field of hardware design and hardware/software co-design.

## 2.1 Optimization problem

The evolutionary algorithms are often used as a heuristics for solving real-world optimization problems. Based on the representation of variables (i.e. whether they are discrete or continuous), the optimization problems can be divided to three subsets – combinatorial, continuous and hybrid optimization problems. The thesis focuses on the continuous optimization problems subset. A **continuous optimization problem** [6] has usually the form of minimizing $f_0(x)$ subject to $f_i(x) \leq b_i, i = 1, \ldots m$, where

- $x = (x_1, \ldots, x_n)$ is the optimization variable,
- $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the objective function,
- $f_i : \mathbb{R}^n \to \mathbb{R}, i = 1, \ldots, m$ are the constraint functions and
- $b_1, \ldots, b_m$ are bounds for the constraints.

There are several classes of optimization problems that are characterized by particular forms of the objective and constraint functions. Some of those classes can be solved numerically in a reasonable time even for a large number of variables. On the other hand, there are some problem classes that cannot be solved exactly for a larger number of variables

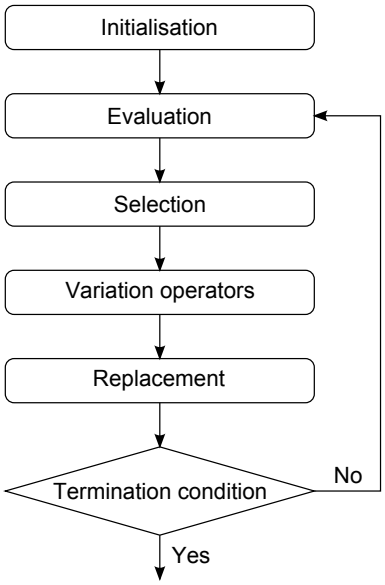


Figure 2.1: Schema of an evolutionary algorithm

in a reasonable (e.g. polynomial) time. These are the domains, where the use of various heuristics such as simulated annealing or evolutionary algorithms has proven to be useful.

There are many classes of optimization problems, where the EAs have been successfully used, for example **Data fitting** [18, 24] or **circuit design** [48, 42]. The following parts contains a brief description of evolutionary algorithm and some of its particular variants.

## 2.2 Evolutionary Algorithms

The evolutionary computation is a research and application field, in which search algorithms inspired in biological evolution are studied and applied. These algorithms are usually population-based and perform a guided random search of particular's problem state space. The evolutionary computation as a whole discipline can be further divided to various sub-areas, such as genetic algorithms, genetic programming, differential evolution, swarm intelligence and grammatical evolution. The following sections briefly describe those parts of the evolutionary computation that are relevant for the thesis.

The evolutionary algorithm (EA) is a general name for a class of algorithms based on Darwin's principle of natural selection. All the variants of evolutionary algorithms share the same basic principles. The overall principle of the EA is depicted in Figure 2.1. Given a population of individuals (represented by a finite genotypes), the selection pressure is applied that causes a natural selection. It means, that the individuals better adapted to the environment (the score is given by a fitness function) have higher chance to survive and produce offspring. Based on their fitness values, the individuals are chosen to produce the next population. Some biologically inspired mechanisms (variation operators), such as recombination and mutation are applied to generate the new offspring which then compete with the old individuals for their place in a new population. This process is repeated until a solution is found or some time limit is reached.

**Variation operators** are used to generate the new individuals from the parents. Therefore the variation operators are responsible for creating the diversity of a population. Thus

they provide the means of how to escape from the local optimum. A genetic operator applied to one individual is called a mutation. The mutation is a stochastic operation that is supposed to cause a random unbiased change (usually atomic) in the genotype. A recombination operator (crossover) is applied to two or more individuals. It can be thought of as an imitation of sexual reproduction in nature. The offspring are generated by combining parts of the parents.

**Survivor selection** (or replacement) takes place after the offspring are generated. For the population size to remain constant, it is necessary to select, which individuals will be retained in the population. Generally the best individuals are chosen to form the next generation.

**Termination condition** is used to ensure the algorithm will stop. A typical termination criterion is based upon the objective function (i.e. when the best-scored individual is sufficient). As this condition does not have to be fulfilled, there can be other conditions ensuring the algorithm will stop. These conditions can, for example, limit the maximum number of generations or terminate the evolution in the case the population is stagnating (i.e. the best individual fitness does not improve).

### 2.2.1 Evolutionary Algorithm Variants

The variants of an evolutionary algorithm usually differ in a representation and genetic operators used. The following paragraphs describe some of the most common variants.

#### Genetic Algorithms

In genetic algorithm (GA) the individuals are usually represented by a fixed-length array of bits. The genetic operators are applied to the genotypes rather than phenotypes. GAs are usually used with bit mutation, crossover and roulette wheel or tournament selection. The GAs have been used in various domains. They are particularly suitable for problems with complex fitness landscape as the algorithm is designed to move the population away from the local optima (by using the genetic operators), where a traditional hill climbing algorithm could get stuck.

#### Evolution Strategies

The evolution strategies (ES) employ a genotype represented by an array of real values. These values encode the parameters of a solution. During the evolution, the selection, mutation and optionally crossover operators are used. The mutation is usually performed by adding a normally distributed random value to each of the real values in the genotype.

The main feature of ES is that the individual does not only represent a point in a search space. It also contains a set of control (or strategy) parameters, for example, a standard deviation of a normal distribution that is used to perform a mutation. These parameters are evolved together with the individual. This process is called self-adaptation.

Currently, two major selection methods are used: $(\mu + \lambda)$ and $(\mu, \lambda)$. The $(\mu + \lambda)$ ES selects the best $\mu$ individuals from the populations of parents and offspring. The $(\mu, \lambda)$ ES chooses the best $\mu$ individuals just from the offspring population.

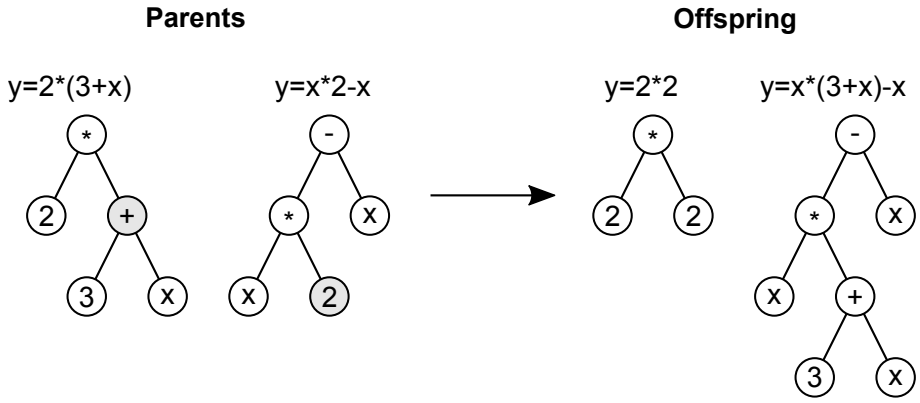


Figure 2.2: Crossover variation operator (crossover points are shown in gray)

### Genetic Programming

Although the genetic algorithms and evolution strategies can be successfully used for problems such as data fitting, there are some problems, where the pure optimization of the parameters is not sufficient. One of these problems is a **symbolic regression**. Basically, the input values are the same as in data fitting – a set of observed and expected values. In contrast to data fitting, the symbolic regression is not just the optimization of parameters of a chosen model. Its purpose is to find the appropriate model and its parameters.

GP may be generally defined as an evolution of programs for the purpose of inductive learning disregarding the genome representation. The modern genetic programming was used for the first time in 1985 by Nichael L. Cramer [12] and then further expanded by John R. Koza in the 90's [27]. The GP was, at first, used for simple problems, but was later used to evolve even human-competitive or patentable inventions [28]. Various GP approaches can be distinguished according to their chromosome representation. These methods are discussed in the following parts. The process of the evolution is basically the same as in GA.

**Tree-based Genetic Programming** (TGP) is the most common approach to genetic programming. The individuals are represented by a tree structure. This structure corresponds to an abstract syntax tree, where the inner nodes represent functions whereas input values and constants are located at leaf nodes. The evaluation is performed by a pre-order or post-order traversal of a tree when the nodes are evaluated using the resulting values of their child nodes. The root therefore holds the final result.

The crossover operation is depicted in Figure 2.2. One node is selected in each of the parents and the offspring are created by swapping the subtrees with the roots at selected nodes. The mutation operator randomly mutates a randomly chosen part of a tree. It can either replace the node with random node of a same type (terminal/nonterminal) or generate a completely new subtree at the point chosen.

**Linear Genetic Programming** operates with the individuals represented by a sequence of instructions in machine code. The basics of this method were proposed by Cramer [12]. The programs represented by the individuals can be executed on a von Neumann architecture composed of a processing unit (containing an arithmetic-logic unit), registers, control unit and memory. The memory holds the data and the instructions. The instructions basically perform an operation upon their operands, which usually originate from

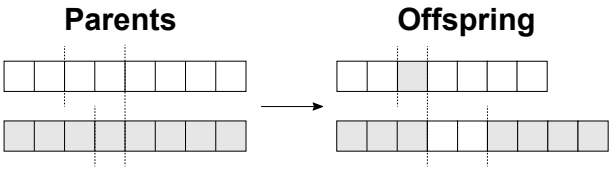


Figure 2.3: Crossover of two LGP individuals

registers and store the result to a chosen register. Branching instruction is usually realised by some kind of the if–else–endif block.

The crossover operation is implemented by a two-point linear crossover in LGP [4]. Figure 2.3 illustrates such a crossover. The mutation operation usually operates at the level of instructions. The instruction undergoing a mutation is chosen randomly and then the operator, source registers or destination register can be replaced by another item from the appropriate sets.

**Cartesian Genetic Programming** (CGP) is one of the most recent variants of GP. CGP was invented by Julian Miller and Peter Thompson in 1999 [33]. It encodes an $n_i$ input, $n_o$ output program using a graph representation. More formally the individual can be thought of as an array of programmable nodes with $n_c$ columns and $n_r$ rows. Each node is described by a function it implements and by the connections of its inputs. Each of the inputs can be connected either to one of the primary inputs of a program or to the output of some preceding node (i.e. no cycles are allowed in the graph). Each node can be, therefore, described by a set of $n_a + 1$ integers, where $n_a$ is the number of node inputs (i.e. the arity of the function). The last integer represents the node function. The whole program is hence encoded by $n_c \cdot n_r \cdot (n_a + 1) + n_o$ integers, where the last $n_o$ integers specify the indices of the nodes whose outputs are taken as program primary outputs.

The mutation operator was the only variation operator in the original CGP version. It operates at the level of individual nodes, where it changes either the function of a node or its inputs. Another type of mutation is the change of output nodes encoded at the end of a chromosome. The search method is based on a simple $(1 + \lambda)$ search strategy, where $\lambda$ is typically less than 10.

CGP has been shown to perform comparably to traditional GP. CGP is popular and quite successful in evolutionary design, optimization and approximation of digital circuits [23, 32, 50]. Many of its variants have emerged, for example Modular CGP (MCGP) [52], Embedded CGP (ECGP) [25] or Self-modifying CGP (SMCGP) [20] [19].

### 2.2.2 Multi-objective Optimization

Many real world problems have more than one objective function that have to be addressed in the fitness function. The objectives of the optimization problem can be even conflicting (i.e. there is some trade-off between them). An example would be maximizing the performance of the engine, while keeping the fuel consumption and emissions low. A multi-objective optimization problem can be formally defined as [13] $min(f_1(x), f_2(x), \ldots, f_k(x))$, where $k$ is the number of objectives and $x \in X$ is an element of a set X, which is the feasible set of decision vectors (with respect to problem constraints including inequalities, equalities and integer constraints). In the case that some objective function $f_j(x)$ is to be maximized, it is equivalent to minimizing the $-f_j(x)$.

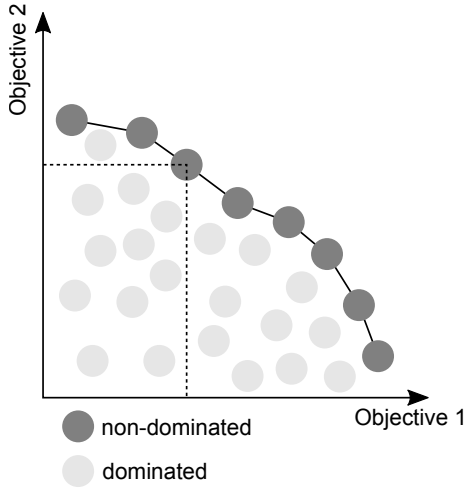


Figure 2.4: Pareto optimal solutions of a function maximizing two objectives. Solutions inside a dashed rectangle are dominated by a solution at its top-right corner.

For non-trivial multi-objective optimization problems, there is usually no global optimum optimizing all the objectives at once. In such case, the comparison of solutions can be based on the idea of Pareto dominance. A solution $x'$ is said to dominate another solution $x''$ if at least on of its objectives i strictly lower and none of the objectives is higher. The solution is considered non-dominated (or Pareto optimal) if there are no other solutions dominating it. Figure 2.4 shows the Pareto optimal solutions of a problem supposed to maximize two objectives. The set of Pareto optimal solutions is often called the Pareto front.

There are several methods that can used to address multiple objectives. These methods can be divided into two separate groups – **a priori** (the objective preference is given or the objectives are scalarized) and **a posteriori** (generating the set of non-dominated solutions after the searching).

### NSGA-II

NSGA-II [14] is one of the most used a posteriori multi-objective optimization methods for evolutionary algorithms. The method works as follows: First, the individuals are evaluated. Then, the Pareto front of the current population is taken and assigned the fitness value (rank) of 1. In the next step, the individuals dominated just by the individuals with fitness of 1 are taken and are assigned the fitness value of 2. This process repeats until all the individuals have the fitness value (rank) assigned.

The crowding distance parameter is then calculated for the individuals. This parameter represents the distance of a particular individual to its neighbours. The larger the average crowding distance in the population is, the better diversity of the population. The idea behind the crowding distance is to support the individuals that differ from the rest of the population to maintain the population diversity.

The parent selection is usually performed by a binary tournament selection. An individual is selected if it has better fitness (rank), than the other one. In case they have the same fitness value, the winner is the individual with greater crowding distance. After the application of variation operators, whole population (old individuals together with new

offspring) is sorted again and then the best individuals are chosen for the next generation. The result of the evolution is a set of non-dominated solutions found during the run.

## 2.3 Hardware/software Co-design

In the 90's the use of electronics and microprocessors in industry and consumer devices such as TVs, game consoles, and mobile phones started to grow rapidly. This caused significant rise of the demand for embedded systems. The embedded systems combine hardware and software part. Therefore, the needs for new methodologies capable of designing such systems emerged. The following sections deal with the description of architectures used in the field of embedded systems and a methodology that transforms the initial specification to a final design ready for manufacturing. As the thesis is focused on helping the designer in the process of a system design by means of evolutionary techniques, the current state-of-the-art in the field of evolutionary hardware design and evolutionary hardware/software co-design is included as well.

### 2.3.1 Universal microprocessor

The basic microprocessor is composed of three main parts – the arithmetic logic unit (ALU), the control unit (CU) and the internal memory (usually in a form of registers). ALU performs the arithmetic and logic operations upon the data provided. Loading of the data and their transfer to the inputs of an ALU is carried out by control unit that controls the data flow inside a processor. Besides the data flow control, the control unit also performs fetching and decoding of the instructions.

From the embedded systems point of view, the use of microprocessors offers several benefits. Most of these benefits come from the microprocessor itself, i.e. a universal computing unit. This allows to employ the same microprocessor for various tasks. Furthermore, the connection with other parts of a system can be made using the existing solutions. This significantly reduces the time needed to design the system. The use of a microprocessor is less expensive than designing an application specific integrated circuit in the case of lower production volumes.

The generic nature of universal processors has, however, its downsides. The main problem is low performance for some class of applications, and on the other hand, high power consumption for other applications. In such cases, the use of application specific integrated circuit or FPGAs can be the only choice despite higher expenses for development.

### 2.3.2 ASIC

Application specific integrated circuits (ASIC) are the opposite of generic architectures. They are designed usually for a particular purpose (e.g. for cell phones) to meet challenging design constraints in terms of power consumption, area and performance. One of the biggest benefits of ASIC is the ability to place all the necessary parts of a system on single chip. Therefore, it is possible to combine digital and analog circuits inside one chip. The downside are high expenses of design and fabrication. The use of ASIC is, therefore, suitable mainly for mass production, where the design costs are distributed among a large number of produced circuits.

As the aforementioned facts imply, the use of custom ASIC technology is suitable mainly for mass production. For lower production volumes, other approaches (e.g. FPGA) are

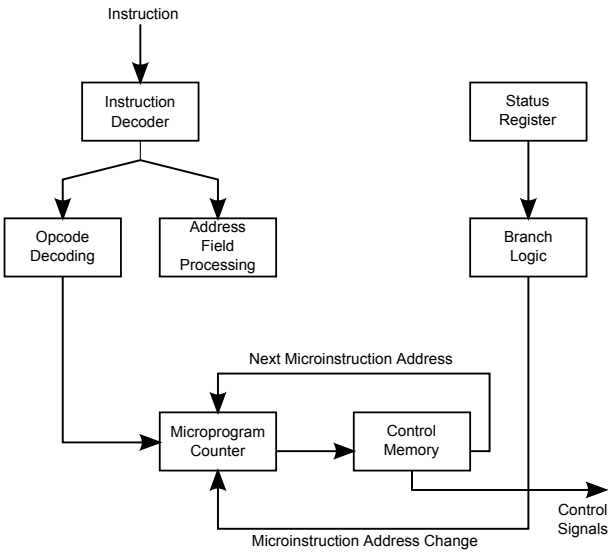


Figure 2.5: Microprogrammed controller scheme

more suitable. FPGAs have traditionally been used for rapid prototyping, but they are currently an important component of many applications. Nevertheless, there are still some applications, where the use of an ASIC is mandatory, either due to lower costs in the case of mass production or due to the requirements given on the final system (e.g. area, power consumption), which cannot be met using other technologies.

### 2.3.3 Microprogram Architectures

Designing a controller for a modern processor employing a complex instruction set is a complicated task. A microprogram architecture is an architecture where the high-level instructions are performed by executing several lower-level instructions (microinstructions) that control the operation of the underlying hardware. A simplified schema of a microprogrammed controller is shown in Figure 2.5. Its operation is basically fetching an instruction, finding the appropriate block of microcode, executing the microinstructions from the block, then fetching next instruction etc.

Encoding of the microinstructions in a memory can in principle be done in two ways. The microcode can be horizontal or vertical. In the case of a horizontal microcode, every item directly contains the values of control signals. In vertical microcode, the microinstruction is encoded using a specific coding system. Therefore less memory is needed, but decoding takes longer.

Microprogram architectures simplify the design of complex systems by representing some parts of the system by a microprogram. Software development is usually less complex than development of a hardware performing the same function. Nowadays, the microprogram architectures are used in high-performance CPUs, where they implement complex instructions. Another usage of microprogram architectures can be, for example, in low-cost digital systems in which it is too expensive or power demanding to employ a processor (e.g. for preprocessing the output of a sensor).

### 2.3.4 High Level Synthesis

High Level Synthesis (HLS) deals with the conversion of system description at high level of abstraction to a description at a lower level while maintaining the constraints given [11]. Regarding the digital system description, several levels of a detail can be used (see [5]). The goal of the synthesis is the conversion of a behavioral description to the RT level. Then, RT level design is transformed to lower levels using well-known methods [31, 17]. Using HLS in the system design brings many advantages, in particular, shortening the design time, supporting algorithmic design verification, possibility of creating multiple implementations etc.

In some cases constraints can be handled as objectives and vice versa. There can be, for example, a requirement constraining maximum area and minimal processing frequency. However, the goal may be to find not just a solution that meets both constraints, but to optimize the area under the frequency constraint. These objectives are often conflicting (e.g. the smaller the area, the lower the frequency is), which implies the existence of a Pareto front of optimal solutions. This can be addressed by using a multi-objective optimization method (see Chapter 2.2.2).

### 2.3.5 HW/SW Codesign Overview

During the design of a system, one of the major steps is to choose, what parts of system will be implemented in hardware and what parts will be performed by software. This decision is usually based on several factors, for example the target costs of the system, the performance requirements or its power consumption. The process of dividing the system to HW and SW part is called **partitioning** and can be performed in numerous ways.

In a traditional design flow, the system is immediately partitioned into HW and SW components. First, the hardware part is synthesized and the result is used as a base for software part design. Then, the software part is compiled and integrated with the hardware part. Then, the simulation can take place and if the results do not satisfy the requirements, the partitioning has to be changed and the whole complicated process has to be repeated.

On the other hand, in the field of HW/SW codesign, the development of HW and SW part is performed concurrently. The flow of a general codesign approach is shown in Figure 2.6. First the system description is converted to a unified hardware/software representation. Using the unified representation, an arbitrary partitioning can be performed dividing the system to HW and SW part. HW part is synthesized and SW part is compiled. These processes take place concurrently. Finally, the HW and SW part are integrated and the resulting system can be simulated. If the design fulfils all the requirements, the codesign process stops. Otherwise a re-partitioning is performed and the process repeats, until a sufficient design is found. It should be noted that the key factor is the unified representation allowing to transfer parts of the system from HW to SW and vice versa.

## 2.4 EA in HW/SW co-design

In the field of HW/SW co-design, various tools have been developed to help designers with particular phases of the design process. These tools frequently employ optimization algorithms. In some cases EAs were integrated to these tools. One of the most developed tools, the MOGAC system, employs a multi-objective genetic algorithm that partitions and schedules embedded system specifications consisting of multiple periodic task graphs [16].

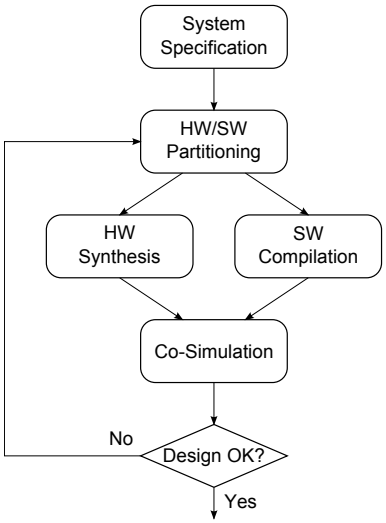


Figure 2.6: HW/SW codesign approach according to [54]

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 [15]. It should be noted that in this case, the algorithm to be implemented by the target system is known and only the co-synthesis process is evolved. Our approach is supposed to evolve even the algorithm itself.

### 2.4.1 Bio-inspired architectures

Biologically inspired engineering applies biological principles known in nature to the study and design of engineering systems. This section discusses the bio-inspired principles utilization in the field of hardware/software co-design.

**MOVE processor** In the context of bio-inspired hardware, the most interesting relevant approach is Tempesti's hardware/software co-evolution of programs and cellular processors [45]. It is built upon the idea that the operation of multi-cellular organisms relies on the specialization of the cells. This implies that the physical structure of a cell is adapted to its function. From the hardware point of view, there is an obvious need of reconfigurable processing elements. Tempesti used the MOVE paradigm [10]. Then, he conducted experiments, whose goal was to transfer parts of the original purely software implementation of a problem solution to a hardware while maintaining area constraints. It has been successfully validated against several benchmarks and the speedup achieved ranged from approximately 1.1 to 5.7.

**Genetic Parallel Programming** Another relevant approach, genetic parallel programming (GPP), has been developed to evolve parallel programs for processors containing multiple ALUs [9]. Based on LGP, it enables to automatically map a problem on parallel resources and evolve a corresponding parallel program. It was shown to successfully solve various classes of problems, for example, symbolic regression, data classification or Fibonacci sequence generation. Although the parameters of the underlying HW (number of ALUs, registers etc.) can be specified, they do not change during the run. Therefore, GPP cannot be considered as an evolutionary HW/SW co-design platform. On the other hand, the designer has the ability to specify the level of parallelism wanted.

# Chapter 3

# Concurrent evolution of HW and SW

We have seen in the previous chapter that evolutionary algorithms (genetic programming in particular) have been employed to design programs as well as hardware. Only to a little extent programs and hardware were optimized together. The main objective of the thesis is to investigate whether and to what extent an evolutionary design based method can evolve a program (SW, for short) concurrently with a programmable computational platform (HW, for short) running the evolved program. From the computer architecture perspective, this problem can be seen as a concurrent development of hardware and software, simply, HW/SW codesign.

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 would be in capturing and preprocessing the data from a sensor in low cost systems. The proposed model of such system consists of three parts: (i) programmable microarchitecture, (ii) program in memory and (iii) environment providing primary inputs and consuming the outputs.

## 3.1 Proposed platform

This section introduces first version of the HW/SW platform developed for experiments with concurrent evolutionary design of HW and SW. After introducing the HW microarchitecture, the programming model and the evolutionary algorithm, results of an initial set of experiments are reported.

### 3.1.1 HW microarchitecture

In order to provide a minimal reasonable HW for our target applications, we employed a well know scheme of a configurable data path that is controlled by means of a microprogram stored in the memory. Figure 3.1 shows that the architecture consists of several interconnected basic components. The whole HW/SW system operates iteratively by executing the instructions of a program (see instruction processing steps in Chapter 2.3.3).

Some of the parts (e.g. registers and modules) can be changed either by a user or by the optimization method, while the others are hard–coded and cannot be changed without modification of the framework's source code. Another noticeable 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 possibility to use the modules simultaneously.

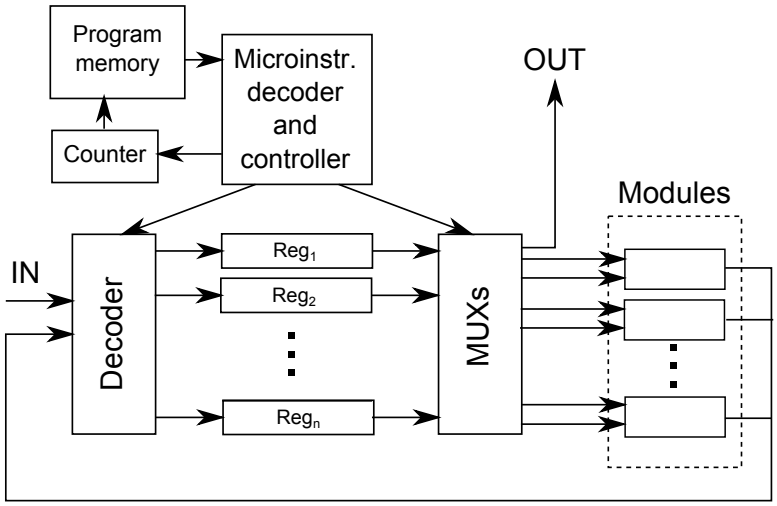


Figure 3.1: HW architecture

Modules can be thought of as black boxes with computational behaviour. Basically, module is specified by several parameters, namely: number of inputs and outputs, area, power consumption, delay and a function specifying module outputs based on the inputs and its internal state. The area, power consumption and delay parameters do not influence module's function, but are used by a framework to optimize the overall architecture.

The architecture definition also contains the registers specification – the register count and bit widths. Various register widths are implemented by their masks. When the register widths are not optimized, 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 to the modules used in the architecture. By default, all the instructions enabled by the modules are supported.

The computational platform from Figure 3.1 can be formalized as follows:

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

Although most of the components of a HW part remain static, the functions $w$ and $u$ can dynamically change the setting of HW during the evolution. Detailed description of the impact of these functions on the fitness of an individual is given in Chapter 3.1.4.

### 3.1.2 SW architecture

The SW architecture specifies the way the program is stored and executed. 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 (i) one or more microinstructions, (ii) corresponding parameters and (iii) inputs and outputs used by them. The instruction block can, therefore, be thought of as a single instruction composed of several microinstructions.

Figure 3.2 shows the format of the microinstruction. There is a mandatory header, which specifies the type of the microinstruction (I/O, branching, module execution). Then there might be a constant. The constant is utilized by branching, register manipulating and I/O instructions. Afterwards, the bytes specifying modules' inputs and outputs 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.

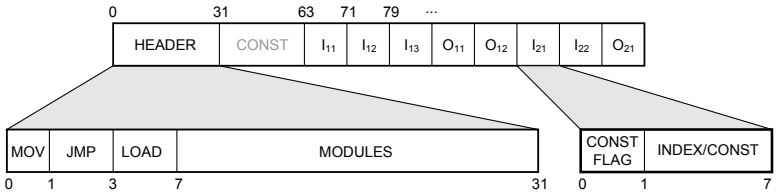


Figure 3.2: Microinstruction format

The program itself is represented by a 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 the case of branching, the program counter is modified to point to a given instruction block and the program execution continues from this point.

### 3.1.3 The model of the environment

The last part needed for successful simulation of a microprogram architecture function is its connection with the environment, where it is going to be used. This can be accomplished by the environment part of the model. The environment can be thought of as a black box providing the inputs for an architecture and consuming its outputs.

The system allows two ways of specifying the environment. The first is the usage of a dataset containing the time series for individual inputs and also the timelines of the expected outputs. 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. Formally, it can be thought of as a Mealy machine, where the symbols of the input alphabet are a combination of action (input/output), signal index and time. From the implementation point of view, this mechanism is represented by two callback functions of an environment class. Enabling these two functions represents a comfortable way to model reactive systems, where the current input value can be based on a system state (that can be changed according to the system interaction) and current logical time.

### 3.1.4 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. In terms of SW there is only one entity to be evolved – the program body itself.

Considering the fact that the SW part of the system is represented by a one–dimensional array of instruction blocks and the programs are executed in a sequential manner, Linear Genetic Programming [7] seems to be the best choice for the framework. This consideration can be supported also by an analysis of available tools of similar nature. GPP [9] and $\mu$GP [43] approaches evolve programs written in imperative languages and subsequently evaluate them against a particular processor (arbitrary microprocessor in the case of $\mu$GP). Both these approaches were successfully used and no serious limitations were observed. Therefore LGP seems to be a good representation for the purpose of the proposed framework. 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.

#### Encoding of HW/SW architecture

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

**HW part** of the architecture represents the usage and bit widths of the registers and the usage of the modules. It is crucial 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 straightforward way. Each register has its bit width specified. When the bit 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 has to be removed, it is just marked as inactive. During the instruction execution the outputs of inactive modules are omitted. The HW architecture is represented as a heterogeneous array containing integer values of register bit widths and binary values of modules deactivation all the aforementioned information.

**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. Therefore, the variation operators are applied at the level of whole instruction blocks and it is not possible to modify the microinstructions inside the instruction block, as the instruction block usually has its meaning as a whole and should not be modified.

#### Generating the initial population

Considering the proposed individual encoding scheme, the program can be generated in a very straightforward manner. It is sufficient to randomly generate the instructions according to the maximal program length specified. The value of the parameter is generated randomly for each instruction block regardless of its type. It is then up to the simulator to resolve the problems (e.g. limiting the jump if it targets outside the program range).

### Computing the fitness function

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

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

It should be noted that speed, area and performance values are not based on a synthesis. For the sake of speed and simplicity, basic circuit parameters are just estimated using simple models in the course of evolution as it is usually done in the evolvable hardware community [49].

The area fitness is estimated according to current bit widths of the registers and the deactivation flags of the modules (i.e. only the active modules are suppossed to occupy an area on a chip). Basically, the area fitness is a normalized reciprocal of a sum of active modules' areas plus registers' areas. The power consumption fitness is evaluated in almost the same way. The speed fitness is a reciprocal value of the processing time.

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

$$f_o = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{1 + d(e_i, o_i)},$$

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 function $d$ is a distance function defining the distance between the expected output and the output generated by the system. The distance function implementation can significantly vary among various problems to be solved. It could be an Euclidean distance, mean squared error aso.

The default functions for the components of the fitness are predefined, but can be changed by the user. As there are four components of a fitness function, the proposed method uses NSGA–II as the method for finding non-dominated solutions. Due to the use of NSGA-II, the tournament selection was chosen as a selection method.

### Crossover

After the parent selection, the crossover takes place. Considering the aforementioned 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.

### Mutation

Because the chromosome is heterogeneous 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 the chromosome).

**HW mutation** can basically influence registers, modules and instruction set. The first of the mutation types modifies the width of a randomly chosen register. The second type modifies the module usage (flips a corresponding bit).

**SW mutation** can modify the program in terms of instructions, their types, parameters, inputs/outputs specification and order. First of all, the microinstruction is randomly selected from a randomly chosen instruction block. Then, one of the types of SW mutation is performed. The first mutation type changes the microinstruction as a whole. The second type of mutation changes only the parameter of the microinstruction. The third type changes the input and output indices of the microinstruction. The last type of mutation doesn't modify the microinstruction itself, but its position in a program.

## 3.2 Basic experimental evaluation

Several basic experiments were carried out to verify the proposed method. This section contains some of them and also a comparison with similar methods. The computational effort (according to [26]) was used as a measure for comparison. It is, however, important to keep in mind that the proposed method has to deal with the concurrent evolution of hardware and software part, whereas available methods are used to develop just the software part.

**Common experiment parameters** If not explicitly stated in particular subsections, the experiments were performed with the following settings: maximum of 20 instr. blocks, population size of 5, max. number of generations 200,000, crossover prob. 0 and mutation prob. 0.7. Two types of modules were used in these basic 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. All operations are performed over 32 bit signed integers.

### 3.2.1 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 distance function, which is the part of the functionality fitness function was defined as number of hits.

After performing several initial runs, some interesting solutions were found. From the hardware point of view, this solution used only one of the two available modules. The registers widths are 6 and 7 bits. Notice the registers have the smallest possible widths to be able to store the last two desired numbers. Therefore, it is the optimal solution in terms of total area used.

Results obtained in this experiment were used to compare the proposed method to other approaches. An average number of evaluations needed to find a functionally correct solution and a success rate (where available) is given in Table 3.1. The comparison shows the proposed method is comparable to state-of-the-art methods.

### 3.2.2 Sextic polynomial

This experiment was chosen mainly to compare the proposed method with existing methods in the field of symbolic regression, because this experiment is one of the widely used benchmarks in the field of GP [36]. The polynomial to be found was $x^6 - 2x^4 + x^2$.

Table 3.1: Comparison of proposed method with other approaches

| | **Average evaluations** | **% Success** |
|---|---|---|
| Proposed method | $1.3 \times 10^5$ | 94 |
| Multi-niche GP [35] | $2 \times 10^5$ | 70 |
| LGP [53] | $2.1 \times 10^5$ | N/A |
| SMCGP [19] | $7.7 \times 10^5$ | 87 |
| Huelsbergen (machine language programs) [21] | $1 \times 10^6$ | N/A |
| Object oriented GP [1] | $2 \times 10^7$ | N/A |

The available modules were chosen so all the operations typically used by other methods were implemented (i.e. 1xADD module and 1xMD module). The environment was defined to process only the first output of a program for a given input. The termination condition was set to stop LGP when all the fitness cases of the training set were satisfied by a candidate program (i.e. it is the number of hits). The allowed error was set to be below 1, as the experiment was conducted using integers. The computational effort estimated from 100 runs is shown in Table 3.2 together with computational efforts of other methods for comparison. The proposed method is comparable to GP and GPP and outperforms some of its variants. However, it is obvious, that the proposed method is outperformed by CGP and ECGP.

Table 3.2: Comparison of experimental results with other methods

| **Method** | **Computational Effort** | **Ratio** |
|---|---|---|
| GPP $\mathcal{M}_{1,2}$ [30] | 5,310,000 | 5.4 |
| GP [26] | 1,440,000 | 1.5 |
| Proposed solution | 990,000 | 1.0 |
| GPP $\mathcal{M}_{8,8}$ [30] | 540,000 | 0.5 |
| CGP [52] | 55,692 | 0.056 |
| ECGP-3 [52] | 54,353 | 0.055 |

## 3.3 Summary

It was shown that the proposed method can evolve microprogrammed architectures capable of solving some of typical problems that were approached by GP in the past. In some cases the proposed method provides better results in terms of computational effort. The method, therefore, seems promising for further exploration.

However, the experiments pointed out some problems of the platform. One of them being the unability to keep a particular value in register without it being overwritten by another value during a program execution. This problem has to be addressed, as overwriting the intermediate results would probably cause much serious issues when evolving more complex programs.

# Chapter 4

# Extended platform: Support for Evaluation of Connected Modules

The experiments described in the previous chapter revealed some limitations of the proposed platform. Those problems were analyzed and appropriate platform extensions were proposed to resolve them. This chapter describes the individual extensions and their validation on a set of nontrivial problems.

## 4.1 Modules topology

The proposed architecture did not allow the evolution to create all useful compositions of modules. As the modules were arranged in parallel to each other, the only possible way of using the outputs of one module in another module was storing the results of one module in registers and passing them to another module in the following instruction. Although this method allowed the modules to use the outputs of previous modules, such connection could be (i) quite resource consuming because of the registers needed to store the intermediate results and (ii) slow because of involving several microinstructions. It was also shown in previous experiments, that there is a problem that values stored in registers may be overwritten by products of randomly created instructions. Therefore, the intermediate results could be overwritten before they could be used in subsequent computations. These problems could be resolved by allowing the architecture to pass the values between the modules directly.

Our solution to this problem is quite simple. The modules have to be ordered and then the inputs of a module $k$ are allowed to be connected only to the outputs of modules $j$ preceding the current module (i.e. $j < k$). The connection of an input can be easily described by an integer like in CGP. However, there are some differences to CGP encoding. First of all, CGP can use an arbitrary number of nodes of each type (limited just by the grid size). Regarding the proposed platform, the number of modules ($m$) is constant. Therefore, they can be labelled and their order could then be described by a permutation $\mu$ of the set $\{1, 2, ..., m\}$. Such permutation could be hardcoded by a designer, but it would significantly limit the search space. Instead, it could be beneficial to let the permutation be a subject to the evolution.

Using permutations encoding in evolutionary techniques is a well-known technique, particularly in the traveling salesman problem (TSP). There are several problems regarding the application of variation operators to chromosomes involving the permutations. First

of all, when using ordinary variation operators, an invalid offspring (permutation) can easily be generated. Such invalid offspring could be disqualified or repaired. However, both approaches proved to be computationally expensive.

Another way is the use of special variation operators that are designed to generate only valid offspring. Many of those operators were proposed and their overview can be found in [29]. The need to deal with special operators could be quite limiting. However, there is yet another possibility devised in [47]. This method uses a specialized encoding of the permutation and allows a direct application of ordinary variation operators. Due to this advantage, this encoding was chosen for the proposed platform.

### 4.1.1 Instructions Related Changes

The changes introduced to the HW part of the architecture required some additional changes in the SW part. Once, more connection options are enabled among the modules, it is necessary to adequately adapt the range of input and output connections during their initial random generation and mutation. In the previous version of the framework, the inputs were allowed to be constants or register indices. To support the interconnections between individual modules the latter one has to be changed, in order to enable the connection of the input either to a specific register or to an output of a module which precedes the current module in the module order specified by the HW part.

The last change imposed by the change of the framework is related to the module outputs. In previous version of the framework, module outputs had to be connected to registers. Considering the possibility to connect the module output directly to another module input, there is no need to store the output of an intermediate module in a register, as it is subsequently processed as an input of next module. It is now possible for a module output to be specified as "no reg", which ensures that the output value can be used by other modules, but will not be stored to a register.

## 4.2 Input Modules

Another issue revealed during the initial experiments was the inability of the framework to process the inputs sequentially by one individual and in parallel by another individual. This limitation had not been too obvious in the previous version of the platform, as there was typically just one module operating at a time. However, after introducing the possibility to create a topology of modules, the need to provide the inputs for all the modules executed becomes clear.

The idea of our solution came from CGP, where the nodes are connected either to other nodes or to primary inputs. As this approach proved useful in CGP, its variation was devised for the proposed platform. The issue was addressed by introducing a new module type – input module. As each input module represents one input, $k$–input system could be modelled by instantiation of $k$ such modules.

It should be noted, that the input modules are not real modules. They are used primarily to resolve the problem with many IN instructions needed in a program. After the solution is found, the execution of these modules can be replaced by inserting actual IN instructions before the instruction in question and connecting the modules to the registers storing the input values.

## 4.3 Experimental Results

Several experiments were carried out to evaluate the proposed extensions. It is important to keep in mind that a comparison with other methods can serve only as a rough assessment of how good the proposed method is, because the proposed method evolves HW and SW part simultaneously and has, therefore, to explore larger search space than the methods evolving just a program for a common processor.

### 4.3.1 Finding the Maximum

This experiment was chosen to verify the ability of the framework to find various (sequential and parallel) different solutions during one run of LGP.

#### Problem Description

The task is to find an architecture calculating the maximum out of 8 input values. There are no further constraints on the number of inputs of the evolved HW part or a processing time. After providing all 8 input values, all successive values will be zeroes and the zero flag of the input module will be set, so the architecture can take an appropriate action. The evolution parameters are listed in Table 4.1. In this experiment a new module type (comparator) was involved. The comparator module (CMP) has two inputs and two outputs and when executed, it sends the smaller value to the first output and the greater one to the second output. The functionality fitness component is defined as the number of correct outputs from 16 semi–randomly generated 8–tuples.

Table 4.1: LGP parameters used for the Maximum task

| Parameter | Value |
|---|---|
| Population size | 50 |
| Max. generation count | 20,000 |
| Crossover probability | 0.05 |
| Mutation probability | 0.7 |
| Max. logical time | 500 |
| Max. program length | 10 |
| Modules used | 8xIN, 8xCMP |

#### Results

After performing 3,000 independent runs the results were analyzed. Out of the total number of 3,000 runs over 62 % have successfully found a solution, with the computational effort of 1,026,114. The evolution was able to find various completely different solutions, including sequential and parallel solutions. The results were sorted by their area fitness and speed fitness. For the sake of clarity, the fitness values were scaled to range $< 0, 100 >$ (the higher the better for both values). The non-dominated solutions are depicted in Figure 4.1.

Figure 4.2 shows one of the solutions evolved. This solution is general, as the inputs are processed in a loop. During each iteration of the loop, two inputs are loaded, compared and the greater of them is passed to another comparator, where it is compared with the value stored in register $r_0$ and the greater of the values is stored back to $r_0$. Therefore, the $r_0$ register contains always the maximum of inputs already processed.

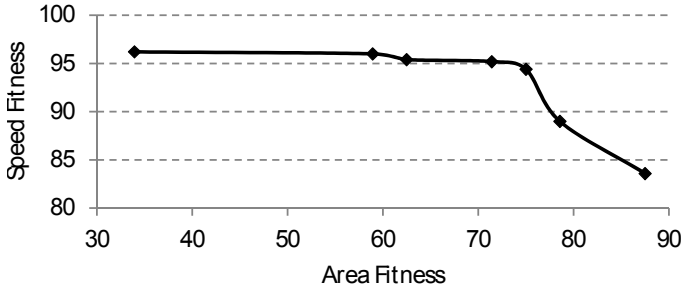


Figure 4.1: The best fully–functional solutions of the maximum experiment

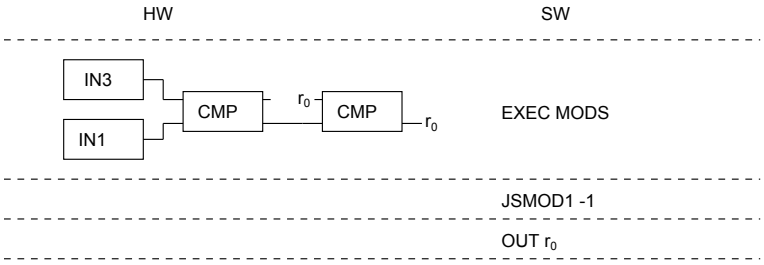


Figure 4.2: An example of a solution of a maximum experiment

### 4.3.2 Parity

This problem was chosen because it is one of typical problems solved using various evolutionary circuit design techniques. Another reason was to find out, whether additional modifications of the platform, potentially speeding up the evaluation could be used.

#### Problem Description

In this experiment, the goal is to find an architecture, which computes parity of the binary inputs provided. The parameters used for the experiment were the same as in the previous case, but the comparator modules were substituted by XOR modules. The functionality fitness component in this case was the number of correct outputs.

#### Results

After performing 3,000 independent runs of LGP the results were evaluated. The computational effort (2,358,430) was surprising as this value is more than twice as large as in the previous experiment, though the problem is quite similar and XOR modules have just one output, whereas the comparator has two outputs.

After some investigation it was concluded that the result is significantly influenced by the definition of the fitness function. Because the XOR function gives just two possible results for each input combination (i.e. 0 or 1), even bad solutions can get quite high fitness. For example, when a candidate solution is a constant (log. 0) producing function, half of the input combinations are evaluated as correct. Therefore the right solution has to have a relatively high fitness value before it is considered better than some of the bad solutions. Despite these problems many correct solutions with various area/speed trade-off were found. The parameters of the best fully–functional solutions are depicted in Figure 4.3.

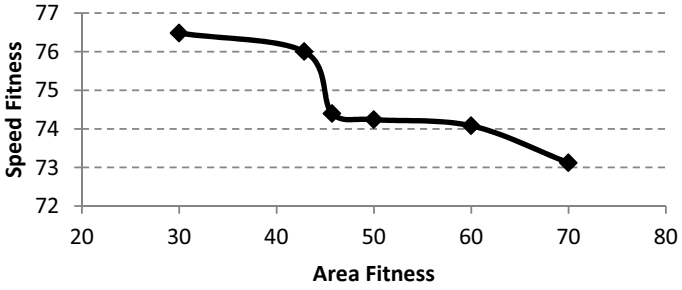


Figure 4.3: The best fully–functional solutions of the parity experiment

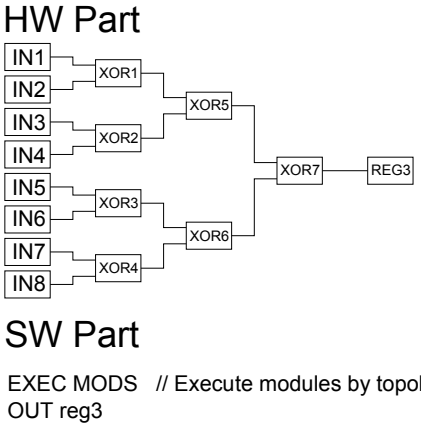


Figure 4.4: Parallel solution for parity

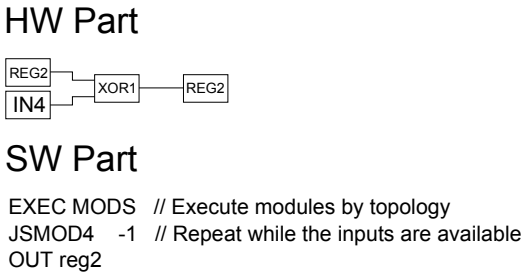


Figure 4.5: Sequential solution for parity

Figure 4.4 shows one of evolved solutions which is fast and resource consuming, but optimized for $n = 8$ and does not process any subsequent inputs. This form of the solution is usually evolved by evolutionary techniques that are not capable of executing loops or creating automatically defined functions (such as standard CGP).

On the other hand, the solution depicted in Figure 4.5 is slower and less expensive. It is also a general solution to the priority problem, as the inputs are loaded in a loop until there are no more inputs available. The computational effort can hardly be compared with other evolutionary techniques as they usually do not use the XOR module, but try to force the evolution to compose the solution from AND, OR and NOT modules.

### 4.3.3 Summary

An important conclusion is that the platform can automatically synthesize multiple implementations, ranging from purely sequential solutions to highly optimized parallel solutions, for a given behavioral specification. This is obviously a fulfilment of one of the goals defined for the thesis, as the designer is provided with multiple solutions and can choose the one best suiting the particular application requirements.

# Chapter 5

# Extended platform: Microinstruction-level modules deactivation

Throughout previous experiments with the framework, low flexibility regarding the strategy in which modules are used was detected. In this chapter, the issue is explained in detail and an appropriate solution is introduced. The proposed extension of the platform is evaluated with two challenging problems: sigmoid function approximation and image filter design.

## 5.1 Modules deactivation

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 excessive number of instructions imposes some problems. First, there is a high probability of destroying vital parts of a program when a mutation changes the instruction type. Second, if large number of modules is used, there is a high probability that some important output of a module stored in a register gets overwritten by another module, which does not have to be used at all.

To address these issues, a modification of the SW part of the chromosome is proposed. This modification adds another property to the 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. These bits can be flipped by a new mutation operator. During the microinstruction execution the modules not utilized by the microinstruction are skipped and their outputs are not available as the inputs for subsequent modules. This way, a particular module can be disabled in context of individual microinstruction. It should be noticed that deactivated (i.e. unneeded) modules do not spoil the values stored in the registers. The downside of this approach is that, in case of deactivation, the inputs of subsequent modules have to be reconnected to other points. This can possibly lead to producing incorrect results.

## 5.2 Case Study 1: Sigmoid Function Approximation

The proposed framework will be evaluated in the task of sigmoid function approximation, which is an important component of hardware implementations of neural networks. In order to reduce a bias of the method, only the inputs and expected outputs will be provided in the training set.

### 5.2.1 Problem description

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.

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. In terms of a bit width, the decision was made to use the fixed-point representation with 6 fractional bits, as according to [46], this precision should be sufficient for implementing a reliable forward operation of a neural network.

### 5.2.2 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 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 5.1. The values of population size, crossover and mutation probabilities were chosen empirically based on previous experiments and several hundreds of runs with different values of these parameters. The functionality fitness was defined as

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

where $e_i$ is $i^{th}$ item from the sequence of expected outputs $(e_1, e_2, ...e_{n_s})$, $o_i$ 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. The termination condition was set to stop the evolution, when all the generated outputs are sufficiently precise (i.e. at least first six fractional bits are correct).

#### Results

100 independent runs were carried out and the solutions found were then examined to assess their quality. The framework was able to find the solution in 19 % of runs. The computational effort needed to find the solution (i.e. 7,520,000) was calculated according to

Table 5.1: 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 |

Table 5.2: Comparison of the computational effort with sextic polynomial symbolic regression

| Problem | Method | Computational effort |
|---|---|---|
| Sextic polynomial | GPP $\mathcal{M}_{1,2}$ [30] | 5,310,000 |
| | GP [26] | 1,440,000 |
| | Original framework [34] | 990,000 |
| | GPP $\mathcal{M}_{8,8}$ [30] | 540,000 |
| Sigmoid approximation | Proposed framework | 7,520,000 |
| | Previous framework version | no solution |

[26]. 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 5.2 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 previous version of the framework, 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.

Afterwards the obtained solutions were examined. One of the solutions found is depicted in Figure 5.1. 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. [56].

Figure 5.2 shows that various non-dominated solutions were discovered. The trade-off between the speed and area fitness is clearly seen.

### 5.2.3 Experiment 2: No multiplication

The framework was quite often able to find a solution utilizing the multiplier module. However, multiplication is quite expensive in terms of the area used. Therefore, the next

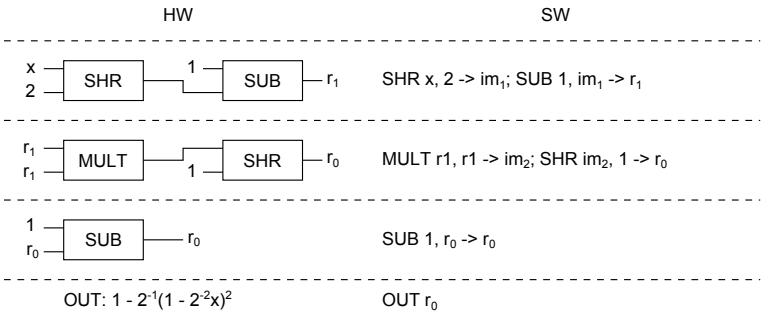


Figure 5.1: Example of evolved solution in Experiment 1

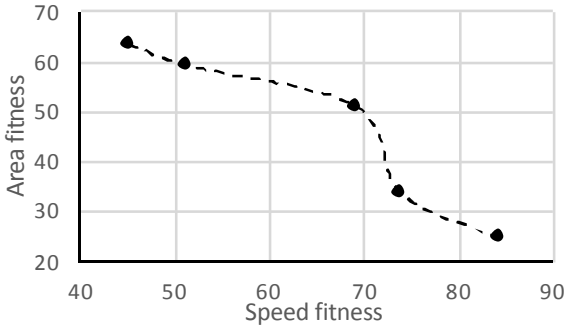


Figure 5.2: Non-dominated solutions for Experiment 1. The values were computed according to [34] and scaled from the $[0; 1]$ interval to $[0; 100]$ interval, where 0 is the worst fitness and 100 is the best (i.e. no modules used or zero execution time).

step was to find a solution that would not need the multiplier. One of such solutions (the so-called PLAN approximation) was proposed in [2]. This solution approximates the sigmoid by 4 linear segments.

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 5.3. In this case, the multiplications by coefficients can be replaced by bit shifts. The corresponding HW implementation would, however, be relatively complex as it uses direct transformation of the inputs to outputs. Finding such a system at once using LGP would probably be nearly impossible. So the goal of our experiment was just to find a piecewise linear approximation of the sigmoid function.

Table 5.3: PLAN approximation of the sigmoid function [2]

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

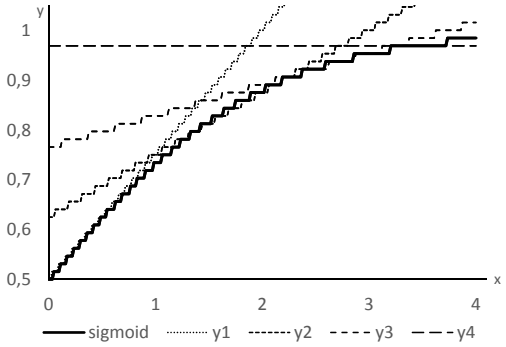


Figure 5.3: Linear approximation A

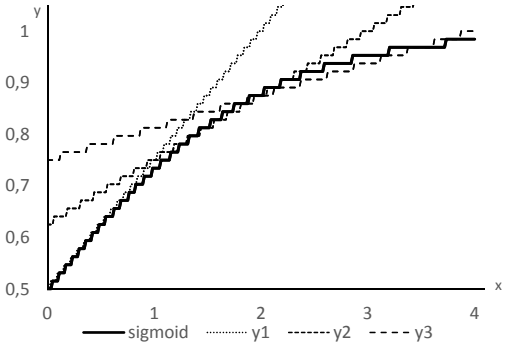


Figure 5.4: Linear approximation B

### Experiment setup

The setup 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. The fitness function was specified as

$$f_o = \frac{1}{1 + \sum_{i=1}^{s} \min_{\forall o \in O} d(o, e_i)},$$

where $s$ is the number of samples, $e_i$ is the expected output for $i^{th}$ sample and $O$ is the set of outputs generated by the individual for particular sample. Therefore, the program is allowed to compute multiple approximations for a given sample, output them and the fitness function will choose the best one. This way, parts of the target sigmoid function can be approximated by different evolved functions.

### Results

Out of 200 independent runs, a suitable solution was found in 2.5 % of cases. Outputs of one of the most precise solutions are depicted in Figure 5.3. The solution (denoted A) 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 the values between 4 and 5, where the gradient of the PLAN approximation is feasible.

Two solution (denoted A and B) are shown in Figurea 5.3 and 5.4. It should be noted, that solution B uses just three linear segments.

Finally, the original PLAN approximation and the two evolved solutions were compared in terms of the average and maximum error. The maximum error is the same but the average

error is smaller for both evolved solutions. Therefore those solutions could be considered superior to the original PLAN approximation in the interval $[0; 4]$.

### 5.2.4 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.

#### 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 [46], 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. The number of 20 Boolean modules is quite high compared to experiments performed with the previous version of the framework. It should, however, assure that the evolution wouldn't be too limited by available resources and confirm that the proposed modules deactivation works as expected. Other parameters of LGP were the same as in previous experiment.

The complete set of all the input combinations was chosen as the input set, because the goal of the experiment was to find a solution giving the correct outputs for all the input combinations. As only Boolean operations were used in the modules, a parallel circuit simulation and compact representation of the truth table with 32 bit integers could be used (according to [37]). The evaluation of the whole training set was then done in one program execution. The outputs were processed in the same manner. The evolution was performed separately for all 6 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). As all possible input combinations are processed at once, it has an interesting side-effect. The training set contains all the combinations of input bits. As none of the inputs is 0 or 1 for all the input combinations, none of the 32bit values representing the inputs can be 0 (all zeroes) or -1 (all ones). The same holds for the outputs. These values can, therefore, be ignored as they only spoil the computation. The modification was made that the Boolean module replaces 0 values on its inputs by -1 values when operating as AND/NAND. When it operates as OR/NOR, the -1 values are replaced by 0 values.

Table 5.4: Logic equations of the solution evolved 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 5.5: Logic equations for the second bit of sig_236p [46]

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

This way, such inputs do not affect the output of a module. 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 [46] (see Tables 5.4 and 5.5). The expressions in the tables use the notation from [46], where the input is of form $x_4x_3.x_2x_1x_0$.

The most important difference 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 an expression in sig_236 could possibly be a mistake or some side-effect of the synthesis (not reported in [46]). 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 [46].

These results have shown that the framework can be used to find the 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.

## 5.3 Case Study 2: Image Filters

During the last two decades, low-cost cameras were employed in many types of electronic devices, e.g. mobile phones, laptops, cars and even watches. As the quality of image preprocessing strongly influences subsequent image processing, the demand for high-quality, low-cost image filters started to grow. Traditionally, linear filters were the most popular. However, there are many areas, where nonlinear filters were proven to give better results [3]. This is strongly influenced by a fact that image signals are usually nonlinear. Another important requirement is the low power consumption of such filters, as they are often used in battery-operated portable devices, where the power consumption is a critical factor. The evolutionary techniques (particularly CGP) have been successfully used to design such filters [40, 41].

### 5.3.1 Problem description

The **impuls noise** is one of the most common types of noise. It is caused mainly by the imperfections of an image sensor. There are two basic types of an impulse noise – the *salt and pepper* noise and the *random-valued shot* noise. The first one causes some pixels to have maximal or minimal value. The latter one causes a random change of some pixels' value. The experiments will be focused of salt-and-pepper type of the noise.

Nonlinear filter operates with the values of pixels in the neighbourhood of the pixel processed. This mechanism is usually implemented using a sliding window function. A sliding window function provides the values of a pixel processed and pixels in a neighbourhood to a nonlinear filter and stores the resulting pixel to the filtered image. This way all the image pixels are processed and the filtered image is produced.

### 5.3.2 Framework settings

As a training image the 128 x 128 pixel picture of Lena was used. The intensity of a noise was chosen to be 5 %. The source image is grayscale, so the bit widths of all registers were limited to 8 bits. As we do not want to bias the evolution towards any particular solution, all the modules used during previous experiments were allowed.Except these modules, the input modules were defined in such way that they would provide the values of pixels $p_0, p_1, \ldots, p_8$ currently contained in the sliding window. The default parameters were used except the maximum number of generations, which was set to 50,000. The fitness function was defined as comparison of filtered image pixels with corresponding pixels of original (uncorrupted) image. The termination criterion was set to stop the evolution when $f_o \geq 0.98$ (where $f_o = 1$ would be an ideal filter).

### 5.3.3 Results

After performing several runs, some solutions fulfilling the termination condition were found. Resulting images were analyzed and some solutions were chosen for analysis. These solutions were then used to filter the 512 x 512 pixel version of the image. The results of an evolved filter (denoted as F1) application are shown in Figure 5.5. After the first application, the noise was apparently reduced, but there were still some corrupted pixels left. After the second iteration the noise is significantly reduced and there is no obvious decrease of image detail. After the analysis, it was discovered, that the solution in fact implements a simple variant of switching-based filter. . This concept has been already proposed in [44]

(a) Original image (b) Corrupted image (c) F1 - first iteration (d) F1 - second iteration

Figure 5.5: (a) Original image, (b) Image corrupted by 5% salt-and-pepper noise, (c) Image filtered by one iteration of F1 filter, (d) Image filtered by 2 iterations of F1 filter

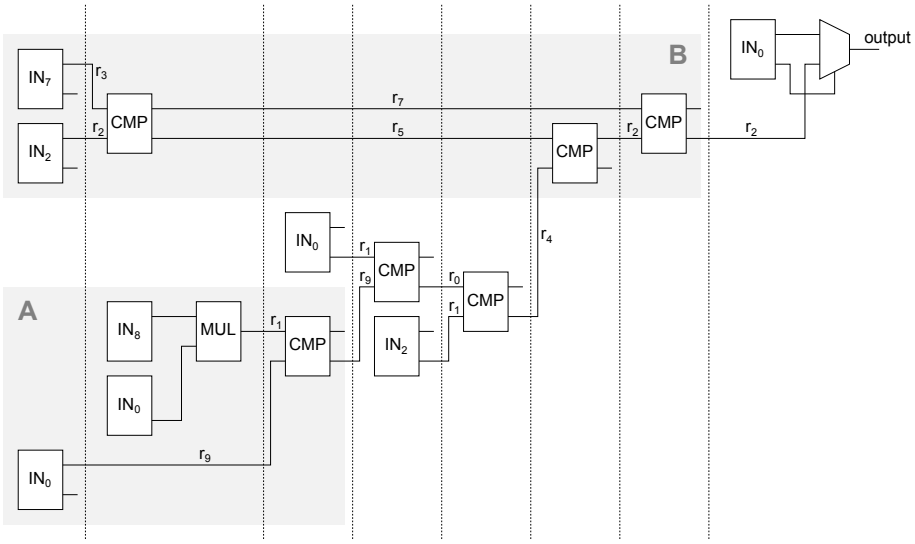


Figure 5.6: Solution employing the two-output input modules. Dashed lines separate individual instructions. MUL represents multiplication, CMP represent comparator.

The discovery of a switching filter led to a decision to modify the input modules. Another output was added to the input module. It outputs 1 if the input is minimal (0) or maximal (255) value and 0 otherwise. Additional runs were performed and some interesting solutions were found. One of them (denoted as F2) is shown in Figure 5.6.

After the examination of the individual, it was observed, it is, in fact, a variation of a median filter restricted to a subset of pixel's neighbourhood. It should be noted that the $r_7$ register holds its value over several instructions. Similar features were not observed in the previous version of the framework. The extensions made to lower the register volatility, therefore, proved beneficial. The application of this filter is shown in Figure 5.7. The noise is significantly reduced even after one iteration. After the second iteration, the noise is almost eliminated and there is no obvious loss of detail.

(a) Original image (b) Corrupted image (c) F2 - first iteration (d) F2 - second iteration

Figure 5.7: (a) Original image, (b) Image corrupted by 5% salt-and-pepper noise, (c) Image filtered by one iteration of F2 filter, (d) Image filtered by 2 iterations of F2 filter

The evolved filters F1 and F2 proved suitable for filtering a low-intensity noise. Then, they were applied to the images corrupted by a higher-intensity noise (50% salt-and-pepper noise). The results are shown in Figure 5.8. The other methods chosen for comparison were (i) ordinary median filter, (ii) adaptive median filter [22] with the maximum size of 5 (denoted AMF5) and (iii) AM-IEPR method proposed by Chan [8]. The methods were

compared in terms of peak signal-to-noise ration (PSNR). The higher the PSNR value, the better image quality. The F1 filter is obivously outperformed by the other methods. The F2 filter produces results comparable to ordinary median filter and AMF5 filter, but is also outperformed by AM-IEPR filter in terms of PSNR and level of detail preserved.

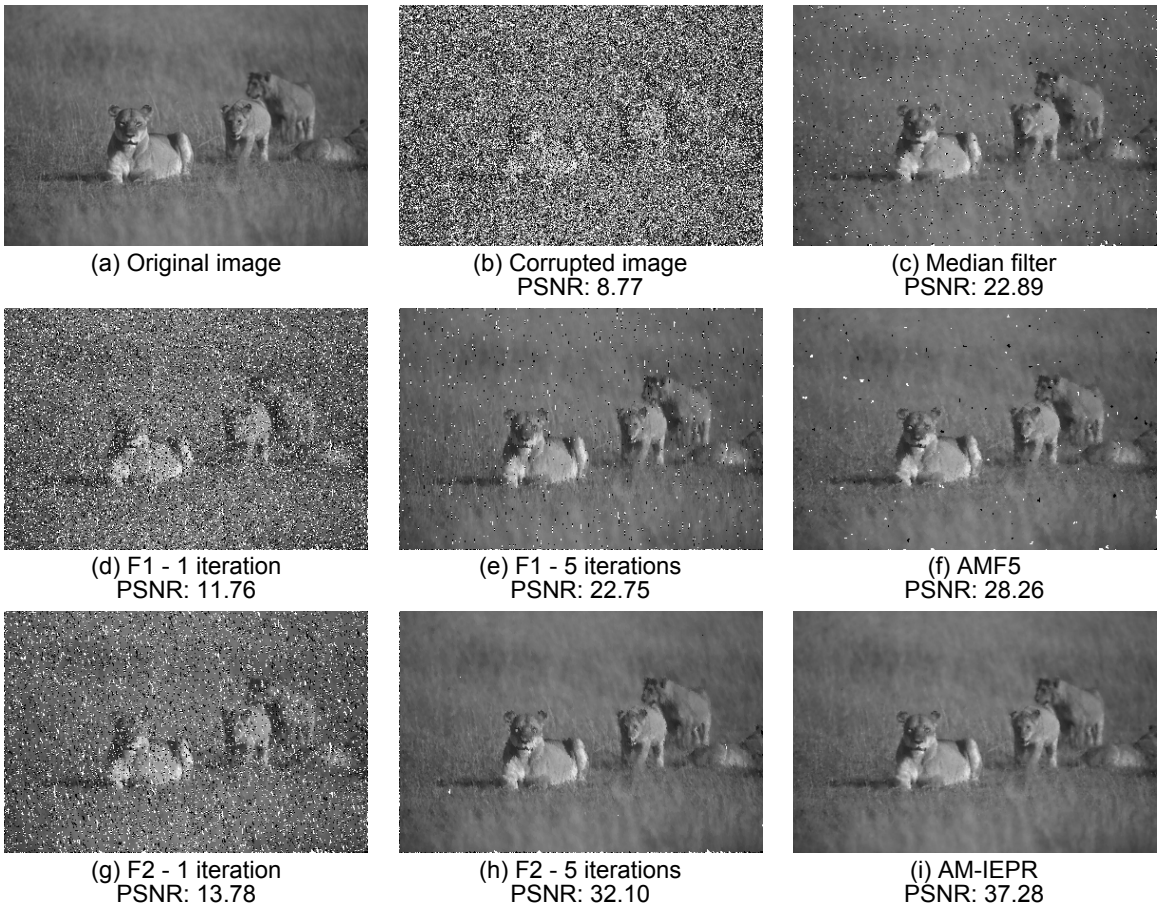


Figure 5.8: Comparison of the filters on an image corrupted by 50% noise

## 5.4 Summary

In this chapter the framework was extended with the possibility to deactivate the modules at a microinstruction level. The proposed extension was evaluated on two non-trivial problems – sigmoid function approximation and non-linear image filters design.

The extended framework was able to evolve variations of well-known sigmoid function approximation algorithms (two sequential and one combinational). In the case of combinational sigmoid approximation, the human designed sig_236 method was reinvented. In the case of non-linear image filters, the framework was able to evolve a switching-based median filter performing comparable to commonly used adaptive median filter even for a high-intensity noise.

# Chapter 6

# Conclusion

The main objective of the thesis was to investigate, whether it is possible to create a framework that will help the user with the design of microprogram architectures using the evolutionary techniques. To the best of our knowledge, there is currently no such framework supporting concurrent evolution of hardware and software. The framework was supposed to provide the user with various implementations showing good trade-off between key system parameters (e.g. area, power consumption). Developing such framework is really a challenging task and designing the complete framework from scratch would be almost impossible. Therefore, we decided to design the framework iteratively, starting with the initial version and extending it based on the results of experiments performed. The ultimate goal was to develop a framework capable of solving real-world problems only on the basis of training data provided.

## 6.1 Goals

In Chapter 1.1, the main objective was split to several sub-goals. The following parts discuss, how and to what extent these sub-goals were fulfilled.

**Goal 1:** The most important point was to specify, how complex the evolved system should be. The field of embedded systems ranges from small systems designed for a simple application to really large systems containing multiple processing units performing really complicated tasks. The proposed framework was, on the contrary, supposed to help with a system design and optimization at a lower level. Therefore, the main target are simple application specific embedded systems – specifically the microprogram architectures. Afterwards, the state-of-the-art in the field of evolutionary algorithms and hardware/software codesign was analyzed. The survey is given in Chapter 2. Based on this survey, the model of the architecture was proposed and the evolutionary method was chosen.

**Goal 2:** Based on the requirements mentioned, the first version of a framework was designed and implemented. The encoding of the individuals was chosen and several variation operators were proposed, that are able to modify the hardware and software parts of the individuals. The components of the fitness function, that represent the key system parameters, were proposed including their default implementations. The detailed description of the framework was given in Chapter 3.1.

**Goal 3:** Several simple experiments were carried out to validate the functionality of the framework. The first experiment proved the ability of the framework to optimize the solution in terms of area. Overall, the proposed method performed comparable to other

methods, but in the case of sextic polynomial, it was clearly outperformed by CGP and ECGP. However, the experiments proved functionality of the framework and also discovered some weak points of the framework were identified. Detailed information about the experiments can be found in Chapter 3.2.

**Goal 4:** There were two important weak points – the volatility of the registers and the inability to process inputs in parallel in a convenient manner. The first problem was addressed by allowing the architecture to make direct connections among modules without the need to pass the values through the registers. This significantly lowered the probability of overwriting the intermediate values before they get processed. The second extension was the support of input modules allowing to process the inputs in parallel. These extensions are described in detail in Chapters 4.1 and 4.2.

The extensions were validated by several experiments. During the experiments, both the extensions proved beneficial. As a result, the framework was able to find several variants of solutions exhibiting various trade-offs. Some of the solutions were sequential, processing the inputs one-by-one, whereas other solutions were parallel, processing all the inputs at once. The solutions found formed a Pareto front allowing the designer to choose the most suitable solution based on particular requirements. The results and examples of the solutions were presented in Chapter 4.3.

Throughout these experiments the problem of registers volatility was encountered again. Another extension was made allowing the modules to be deactivated at the level of microinstructions. This extension proved useful in lowering the probability of spoiling the register content.

**Goal 5:** Finally, the framework was validated on more complex problems. The first set of the experiments was focused on effective implementation of sigmoid function approximation (see Chapter 5.2). Various sequential implementations of sigmoid approximation were evolved, some of them realizing the well-known piecewise second-order approximation or PLAN approximation. The last variant of the approximation was purely combinational and was, in fact, a reinvention of the sig_236 approximation proposed in [46].

The second set of experiments was supposed to evolve an image filter reducing salt-and-pepper impulse noise (see Chapter 5.3). In this case, the framework was able to evolve the concept of switching-based filter proposed in [44]. After minor modifications, the variation of switching-based median filter was automatically generated that performed comparable to the filters commonly used.

### Summary

The proposed framework proved its ability to evolve a hardware platform concurrently with a program in such way, that the user is provided with multiple implementations exhibiting various trade-offs between key system parameters. The framework was successfully used to solve a set of problems of varying complexity. It was able to evolve solutions of real-world problems and even to reinvent some of the concepts previously published as new inventions.

There are still many challenges not addressed by this thesis. The most important one is eliminating the inherent scalability problem if the approach should be used to solve more complex problems. But it is a general problem of the problem solving methods based on evolutionary design principles.

# Bibliography


[1] Agapitos, A.; Lucas, S. M.: *Learning Recursive Functions with Object Oriented Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2006. ISBN 978-3-540-33144-5. pp. 166–177.

[2] Amin, H.; Curtis, K. M.; Hayes-Gill, B. R.: Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings - Circuits, Devices and Systems*. vol. 144, no. 6. 1997: pp. 313–317.

[3] Astola, J.: *Nonlinear Filters for Image Processing*. Bellingham, WA, USA: Society of Photo-Optical Instrumentation Engineers (SPIE). 1999. ISBN 0819430331.

[4] Banzhaf, W.; Francone, F. D.; Keller, R. E.; et al.: *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. 1998. ISBN 1-55860-510-X.

[5] Bell, G. C.; Newell, A.: *Computer Structures: Readings and Examples*. McGraw-Hill. 1971. ISBN 978-0-07-004357-2.

[6] Boyd, S.; Vandenberghe, L.: *Convex Optimization*. New York, NY, USA: Cambridge University Press. 2004. ISBN 0521833787.

[7] Brameier, M. F.; Banzhaf, W.: *Linear Genetic Programming*. Springer Publishing Company, Incorporated. first edition. 2010. ISBN 1441940480, 9781441940483.

[8] Chan, R. H.; Ho, C.-W.; Nikolova, M.: Salt-and-pepper Noise Removal by Median-type Noise Detectors and Detail-preserving Regularization. *Trans. Img. Proc.*. vol. 14, no. 10. October 2005: pp. 1479–1485. ISSN 1057-7149.

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

[10] Corporaal, H.; Arnold, M.: Using Transport Triggered Architectures for Embedded Processor Design. *Integr. Comput.-Aided Eng.*. vol. 5, no. 1. January 1998: pp. 19–38. ISSN 1069-2509.

[11] Coussy, P.; Morawiec, A.: *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer. first edition. 2008. ISBN 978-1-40-208587-1.

[12] Cramer, N. L.: A Representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc.. 1985. ISBN 0-8058-0426-9. pp. 183–187.

[13] Deb, K.; Kalyanmoy, D.: *Multi-Objective Optimization Using Evolutionary Algorithms*. New York, NY, USA: John Wiley & Sons, Inc.. 2001. ISBN 047187339X.

[14] Deb, K.; Pratap, A.; Agarwal, S.; et al.: A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*. vol. 6, no. 2. Apr 2002: pp. 182–197. ISSN 1089-778X. doi:10.1109/4235.996017.

[15] Deniziak, S.; Gorski, A.: Hardware/Software Co-synthesis of Distributed Embedded Systems Using Genetic Programming. In *Evolvable Systems: From Biology to Hardware*, *LNCS*, vol. 5216. Springer. 2008. pp. 83–93.

[16] Dick, R. P.; Jha, N. K.: 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. 1998: pp. 920–935.

[17] Elliott, J.: *Understanding behavioral synthesis : a practical guide to high-level design*. Boston: Kluwer Academic. 1999. ISBN 9780792385424.

[18] Gulsen, M.; Smith, A.; Tate, D.: A genetic algorithm approach to curve fitting. vol. 33. 07 1995: pp. 1911–1923.

[19] Harding, S.; Miller, J. F.; Banzhaf, W.: Self modifying Cartesian Genetic Programming: Parity. In *2009 IEEE Congress on Evolutionary Computation*. May 2009. ISSN 1089-778X. pp. 285–292. doi:10.1109/CEC.2009.4982960.

[20] Harding, S.; Miller, J. F.; Banzhaf, W.: Developments in Cartesian Genetic Programming: Self-modifying CGP. *Genetic Programming and Evolvable Machines*. vol. 11, no. 3-4. September 2010: pp. 397–439. ISSN 1389-2576.

[21] Huelsbergen, L.: Learning recursive sequences via evolution of machine-language programs.

[22] Hwang, H.; Haddad, R. A.: Adaptive Median Filters: New Algorithms and Results. *Trans. Img. Proc.*. vol. 4, no. 4. April 1995: pp. 499–502. ISSN 1057-7149. doi:10.1109/83.370679.

[23] Kalganova, T.; Miller, J.: Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*. 1999. pp. 54–63. doi:10.1109/EH.1999.785435.

[24] Karr, C.; Stanley, D.; Scheiner, B.; et al.: *Genetic algorithm applied to least squares curve fitting*. Report of investigations. U.S. Dept. of the Interior, Bureau of Mines. 1991.

[25] Kaufmann, P.; Platzner, M.: Advanced Techniques for the Creation and Propagation of Modules in Cartesian Genetic Programming. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*. GECCO '08. New York, NY, USA: ACM. 2008. ISBN 978-1-60558-130-9. pp. 1219–1226.

[26] Koza, J.: *Genetic programming II: automatic discovery of reusable programs*. Complex adaptive systems. MIT Press. 1994. ISBN 9780262111898.

[27] Koza, J.: *On the programming of computers by means of natural selection*. A Bradford book. MIT Press. 1996. ISBN 9780262111706.

[28] Koza, J. R.: Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*. vol. 11. September 2010: pp. 251–284. ISSN 1389-2576.

[29] Larrañaga, P.; Kuijpers, C. M. H.; Murga, R. H.; et al.: Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artif. Intell. Rev.*. vol. 13, no. 2. April 1999: pp. 129–170. ISSN 0269-2821.

[30] Leung, K. S.; Lee, K. H.; Cheang, S. M.: Parallel programs are more evolvable than sequential programs. In *Proceedings of the 6th European conference on Genetic programming*. EuroGP'03. Berlin, Heidelberg: Springer-Verlag. 2003. ISBN 3-540-00971-X. pp. 107–118.

[31] McFarland, M.; Parker, A.; Camposano, R.: The high-level synthesis of digital systems. *Proceedings of the IEEE*. vol. 78, no. 2. feb 1990: pp. 301 –318.

[32] Miller, J. F.: Digital filter design at gate-level using evolutionary algorithms. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*. Morgan Kaufmann Publishers Inc.. 1999. pp. 1127–1134.

[33] Miller, J. F.: An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. GECCO'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. 1999. ISBN 1-55860-611-4. pp. 1135–1142.

[34] Minarik, M.; Sekanina, L.: Concurrent Evolution of Hardware and Software for Application-Specific Microprogrammed Systems. In *2013 IEEE International Conference on Evolvable Systems (ICES)*. Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE Computational Intelligence Society. 2013. ISBN 978-1-4673-5869-9. pp. 43–50.

[35] Nishiguchi, M.; Fujimoto, Y.: Evolution of recursive programs with multi-niche genetic programming (mnGP). In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*. May 1998. pp. 247–252. doi:10.1109/ICEC.1998.699720.

[36] Poli, R.: New Ideas in Optimization. chapter Parallel Distributed Genetic Programming. Maidenhead, UK, England: McGraw-Hill Ltd., UK. 1999. ISBN 0-07-709506-5. pp. 403–432.

[37] Poli, R.; Langdon, W. B.: Advances in Genetic Programming. chapter Sub-machine-code Genetic Programming. Cambridge, MA, USA: MIT Press. 1999. ISBN 0-262-19423-6. pp. 301–323.

[38] Poli, R.; Langdon, W. B.; McPhee, N. F.: *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd. 2008. ISBN 1409200736, 9781409200734.

[39] Samuel, A. L.: AI, Where It Has Been and Where It Is Going. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*. 1983. pp. 1152–1157.

[40] Sekanina, L.: *Image Filter Design with Evolvable Hardware*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2002. ISBN 978-3-540-46004-6. pp. 255–266.

[41] Sekanina, L.: *Evolvable Components: From Theory to Hardware Implementations*. SpringerVerlag. 2004. ISBN 3540403779.

[42] Shahookar, K.; Esbensen, H.; Mazumder, P.: Genetic Algorithms for VLSI Design, Layout &Amp; Test Automation. chapter Standard Cell and Macro Cell Placement. Upper Saddle River, NJ, USA: Prentice Hall PTR. 1999. ISBN 0-13-011566-5. pp. 69–106.

[43] Squillero, G.: MicroGP—An Evolutionary Assembly Program Generator. *Genetic Programming and Evolvable Machines*. vol. 6, no. 3. Sep 2005: pp. 247–263. ISSN 1573-7632.

[44] Sun, T.; Neuvo, Y.: Detail-preserving Median Based Filters in Image Processing. *Pattern Recogn. Lett.*. vol. 15, no. 4. April 1994: pp. 341–347. ISSN 0167-8655.

[45] 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)*. IEEE Computer Society. 2006. pp. 129–136.

[46] Tommiska, M. T.: Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings - Computers and Digital Techniques*. vol. 150, no. 6. 2003: pp. 403–411.

[47] Üçoluk, G.: Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation. *Intelligent Automation & Soft Computing*. vol. 8, no. 3. 2002: pp. 265–272. doi:10.1080/10798587.2000.10642829.

[48] Varanelli, J. M.; Cohoon, J. P.: Population-Oriented Simulated Annealing: A Genetic/Thermodynamic Hybrid Approach to Optimization. In *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. 1995. ISBN 1-55860-370-0. pp. 174–183.

[49] Vasicek, Z.; Sekanina, L.: *Circuit Approximation Using Single- and Multi-objective Cartesian GP*. Cham: Springer International Publishing. 2015. ISBN 978-3-319-16501-1. pp. 217–229.

[50] Vassilev, V. K.; Job, D.; Miller, J. F.: Towards the automatic design of more efficient digital circuits. In *Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware*. 2000. pp. 151–160. doi:10.1109/EH.2000.869353.

[51] Volder, J. E.: The CORDIC Trigonometric Computing Technique. *Electronic Computers, IRE Transactions on*. vol. EC-8, no. 3. sept. 1959: pp. 330 –334.

[52] Walker, J. A.; Miller, J. F.: The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*. vol. 12, no. 4. Aug 2008: pp. 397–417. ISSN 1089-778X. doi:10.1109/TEVC.2007.903549.

[53] Wilson, G.; Heywood, M.: Learning Recursive Programs with Cooperative Coevolution of Genetic Code Mapping and Genotype. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. GECCO '07. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-697-4. pp. 1053–1061.

[54] Wolf, W.; Madsen, J.: Embedded systems education for the future. *Proceedings of the IEEE*. vol. 88, no. 1. Jan 2000: pp. 23–30. ISSN 0018-9219. doi:10.1109/5.811598.

[55] Wortmann, F.; Flüchter, K.: Internet of things. *Business & Information Systems Engineering*. vol. 57, no. 3. 2015: pp. 221–224.

[56] Zhang, M.; Vassiliadis, S.; Delgado-Frias, J. G.: Sigmoid generators for neural computing using piecewise approximations. *IEEE Transactions on Computers*. vol. 45, no. 9. 1996: pp. 1045–1049.

# List of Publications

- Minařík, M.; Sekanina, L.: Evolution of Iterative Formulas Using Cartesian Genetic Programming. *Lecture Notes in Computer Science*. vol. 2011, no. 6881. 2011: pp. 11–20. ISSN 0302-9743.
- Minařík, M.; Sekanina, L.: Concurrent Evolution of Hardware and Software for Application-Specific Microprogrammed Systems. In *2013 IEEE International Conference on Evolvable Systems (ICES)*. Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE Computational Intelligence Society. 2013. ISBN 978-1-4673-5869-9. pp. 43–50.
- Minařík, M.; Sekanina, L.: Exploring the Search Space of Hardware / Software Embedded Systems by Means of GP. In *Genetic Programming, 17th European Conference, EuroGP 2014*. LNCS 8599. Springer Verlag. 2014. ISBN 978-3-662-44302-6. pp. 112–123.
- Minařík, M.; Sekanina, L.: On Evolutionary Approximation of Sigmoid Function for HW/SW Embedded Systems. In *20th European Conference on Genetic Programming, EuroGP 2017*. LNCS 10196. Springer International Publishing. 2017. ISBN 978-3-319-55696-3. pp. 343–358.

# Curriculum Vitae

## Personal Data

Miloš Minařík
U Stadionu 548
595 01 Velká Bíteš

e-mail: minam.net@email.cz

## Education

| | |
|---|---|
| 09/1992–06/2001 | ZŠ Velká Bíteš |
| 09/2001–06/2005 | SPŠSS a VOŠT Brno, Sokolská |
| 09/2005–06/2008 | Brno University of Technology - Bachelor's Degree |
| 09/2008–06/2010 | Brno University of Technology - Master's Degree |
| 09/2010–present | Brno University of Technology - Ph.D. Degree (in progress) |

## Work Experience

| | |
|---|---|
| 2007–present | Self-employed |

## Competencies

Czech, English

## Skills

| | |
|---|---|
| Op. Systems | Windows, Linux |
| Languages | C, C++, Python, Perl, Pascal, Ruby, R |
| Databases | MS SQL, PostgreSQL, ElasticSearch, OrientDB, Neo4j |

## Interests

Sport, Music, Movies, Guitar, Travelling

Brno, 08/31/2017