**Noname manuscript No.**
(will be inserted by the editor)

# Comparing Three Online Evolvable Hardware Implementations of a Classification System

**Oscar Garnica · Kyrre Glette · Jim Torresen**

Received: date / Accepted: date

**Abstract** In this paper, we present three implementations of an online evolvable hardware classifier of sonar signals on a 28 nm process technology FPGA, and compare their features using the most relevant metrics in the design of hardware: area, timing, power consumption, energy consumption, and performance. The three implementations are: one full-hardware implementation in which all the modules of the evolvable hardware system, the evaluation module and the Evolutionary Algorithm have been implemented on the ZedBoard™ Zynq® Evaluation Kit (XC7-Z020 ELQ484-1); and two hardware/software implementations in which the Evolutionary Algorithm has been implemented in software and run on two different processors: Zynq® XC7-Z020 and MicroBlaze™. Additionally, each processor-based im-

O. Garnica
Dpto. Arquitectura de Computadores, Universidad Complutense de Madrid, C/ Profesor José García Santesmases, 28040-Madrid, Spain
E-mail: ogarnica@ucm.es

Kyrre Glette · Jim Torresen
Department of Informatics, University of Oslo, Oslo 0316, Norway

plementation has been tested at several processor speeds. The results prove that the full-hardware implementation always performs better than the hardware/software implementations by a considerable margin: up to $\times 7.74$ faster that MicroBlaze, between $\times 1.39$ and $\times 2.11$ faster that Zynq, and $\times 0.198$ lower power consumption. However, the hardware/software implementations have the advantage of being more flexible for testing different options during the design phase. These figures can be used as a guideline to determine the best use for each kind of implementation.

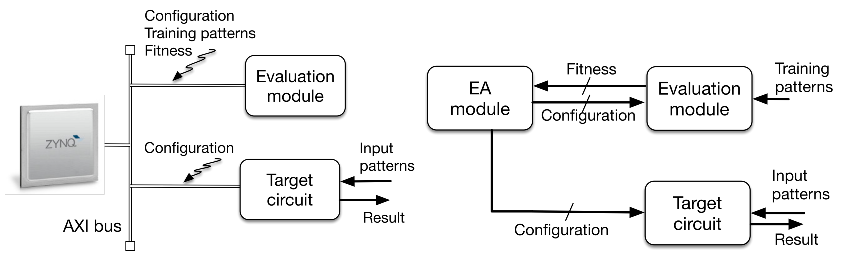
**Keywords** Evolutionary algorithms, evolvable hardware, classifier system, field programmable gate arrays

**CR Subject Classification** Evolvable hardware · Reconfigurable logic and FPGAs · Evolutionary Algorithms and Classification and regression trees

## 1 Introduction

Evolvable Hardware (EHW) arises from the combination of reconfigurable hardware with Evolutionary Algorithms (EA) [28,37,25]. In EHW, the EAs can be used to design the reconfigurable hardware in two distinctive ways:

– ***During the design phase***, EAs can be used as heuristics to explore the design space of the target system, for example in [24,33]. In this case, either intrinsic evolution, in which new designs are run on actual hardware, or extrinsic evolution, in which new designs are simulated on a computer, are possible [14].
– ***During circuit operation***, EAs can be used as a heuristic to find the best circuit configuration at runtime. This new configuration will modify the circuit behavior and, in this way, it will adapt the circuit to an environment that has departed



(a) Hardware/software implementation. (b) Full-hardware implementation.

**Fig. 1** EHW approaches.

from the specifications that were defined during the design phase. An example can be found in [9]. In this case, hardware adaptation is exclusively run online while the actual hardware operates in a real physical environment [14].

If we restrict our study of the EHW to the latter approach, it has the following features: (1) the EA runs onchip, that is, on the same chip as the EHW target system; (2) the EHW adapts itself online, i.e. the EA modifies some design features in runtime; and (3) it applies the notion of intrinsic evolution, i.e. the evaluation of a potential candidate is performed on the actual hardware. It can be implemented with EA running in software or being implemented in hardware, as illustrated in Fig. 1(a) and Fig. 1(b) respectively.

Commercial FPGAs are a good technology candidate for this kind of system thanks to their availability at low cost and their reconfiguration capabilities. Consequently, they have been used as a development platform for this kind of system in almost all earlier implementations.

Most of the papers in the literature use the above approaches, but there is no a systematic comparison between them using the *same* testbench. The aim of this article is to compare different hardware approaches to the implementation of an online EHW system using modern Intellectual Property (IP) processors and current process technology FPGAs, and draw some conclusions about the most appropriate hardware implementation for each use case.

The online EHW testbench is a classifier system applied to a sonar target recognition task. Glette *et al.* [10] implemented this system using the hardware/software implementation and a PowerPC 405 hard processor core on a Xilinx XC2VP30 FPGA. Nevertheless, we are targeting the evaluation of the different hardware approaches to implement the evolutive system and not the accuracy of the system itself, that was covered in [9,11,16]. So, we have used the classifier as a soft IP core and synthesized it to work on the Zynq® XC7-Z020 target FPGA to obtain a fair comparison on a cutting-edge 28 nm process technology FPGA. We also have implemented the EHW system using three alternatives. Two implementations have been designed using a SoC with two different processors: Zynq® XC7-Z020 PS7 5.4 built-in dual-core ARM® Cortex™-A9 processor system and MicroBlaze™ 9.3 soft-core processor. In the third implementation, an IP hardware module implements the EA, including a hardware random number generator. Also, we compare the key metrics in the design of hardware: area, timing, power consumption, energy consumption, and execution time.

This paper is structured as follows. Section 2 presents the previous work on EA implementation of EHW. Section 3 introduces the target application and Section 4 describes the EA. Section 5 presents the EHW architecture. In Section 6 we describe the methodology, in Section 7 we present the experimental results, and finally Section 8 is devoted to the paper's conclusions.

## 2 Previous work

As stated in Section 1 there are two alternative architectures to implementing the EA of an EHW on FPGAs. On the one hand, the software architecture uses a processor (ARM, MicroBlaze, Zynq, PowerPC, etc.) to execute the EA software. On the other hand, the hardware alternative uses a custom hardware module responsible for running the EA.

There are also two main approaches to implementing the reconfigurable capabilities of the EHW on a FPGA [21]. On the one hand, the use of Virtual Reconfigurable Circuits (VRC) in which a virtual circuit is defined on top of the reconfigurable FPGA structures. The virtual circuit configuration can change without dynamically reconfiguring the FPGA. Usually, this approach uses the Cartesian Genetic Programming (CGP) to find the virtual circuit configuration that maximizes the value of a fitness function. Numerous studies combine this approach with any of the EA architectures to design medium-high complexity circuits. Among the most recent examples are [8] than runs the EA in a MicroBlaze processor, [35,34] in a custom hardware, or [32] in a PowerPC.

On the other hand, Xilinx FPGAs have the feature of self-reconfiguring through the Internal Configuration Access Port (ICAP) [2]. This feature allows Dynamic Partial Reconfiguration (DPR) of the partitions in the design so that the circuit modules can be evolved on-line and the result of the evolution can be implemented in the circuit in real-time. Again this approach can be combined with any of the EA architectures. For example, it has been used to modifying the structure of the nodes of a VRC for small combinational circuits running the EA on a PowerPC [5,4], or for the reconfiguration of the category decision modules of a classifier using a PowerPC [30].

In [21], Salvador presents a review of the most remarkable works in EHW design up to date. He states that [20,7] are the more sophisticated and advanced applications. In [20], Mora *et al.* present the DPR implementation of an image filter using a systolic array on a SoC. The system is implemented on a Xilinx Virtex-5 LX110T FPGA and uses a software architecture with an Evolution Stratergy, (1 + 1)-ES, executed for 32768 generations on a MicroBlaze processor. The implementation of the system requires 2688 slices excluding the resources utilized by the MicroBlaze processor.

Dobai *et al.* [7] present a low-level architecture that merges the virtual and native reconfiguration approaches for the definition of a candidate solution. The system is implemented on an XC7Z020 Zynq-7000 All Programmable SoC device card using a software architecture and tested with an image filter. The design requires 3899 total LUTs and employs a (1 + $\lambda$)-ES with $\lambda$ being the number of offspring, $\lambda \in [1, 8]$, that runs on an ARM Cortex-A9 processor. The length of the chromosome is 388 bits, and they run $10^5$ generations.

In [6] the authors analyze the Zynq-7000 All Programmable SoC platform, which contains an XC7Z020-1CLG484CES device, to design EHW. They tested the board using an image filter and applying symbolic regression by CGP. They use an (1 + 4)-ES using VRC and DPR. The article is not focused on the area overhead nor the power consumption of the different approaches but the time needed to evaluate a given number of generations. The DPR implementation of the system requires 6650 slices while the VRC implementation requires 1290 slices and 1084 FF slices. They run 100 generations, and the chromosome's length is 384 bits.

So, to our best knowledge, there is no earlier paper devoted to implementing the same EHW using different architectures to run the EA on current FPGA technology and compare them regarding the most relevant metrics in the design of hardware.

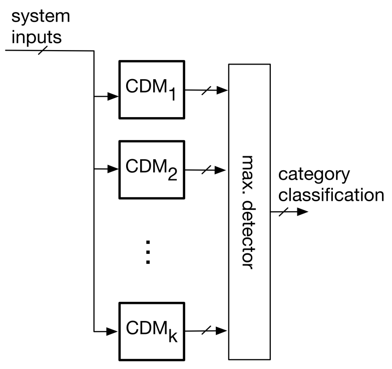
Our testbench is a classifier system used in several EHW applications [10,30,11, 29]. The system complexity[1] is in line with the average complexity of EHW systems presented above. The selected application is a sonar target recognition task widely used as Neural Networks or classifier system benchmark [3,12,1]. The number of generations, the types of mutation operators and the chromosome's length are also aligned with the average applications used in EHW as presented above. Applications that require more generations will not qualitative change the conclusions of the comparisons because the number of generations affect at the features of the three implementations in a similar way. Regarding the mutation operator, the usage of more complex operators is not very common in EHW and, even in that case, they are entirely feasible in hardware at the clock frequencies we use in this work. Again, this will not change the results significantly. On the other hand, neither real-time constraints nor power and energy consumptions are an issue for the selected application, but applications exist where classifier systems need to be fast and use low power [18,17]. Furthermore, a fast evolution time is a requirement in most of the EHW applications, and one of the niches of EHW are evolvable embedded systems where power and energy consumptions are relevant [25,14,18]. For such reasons, we will measure these metrics. The fact that the sonar recognition is aimed neither real-time nor low power and energy consumptions does not invalidate the results because the paper focuses on comparing the architectural alternatives in the implementation of the EA.

[1] See Section 5 for a detailed description of the circuit.

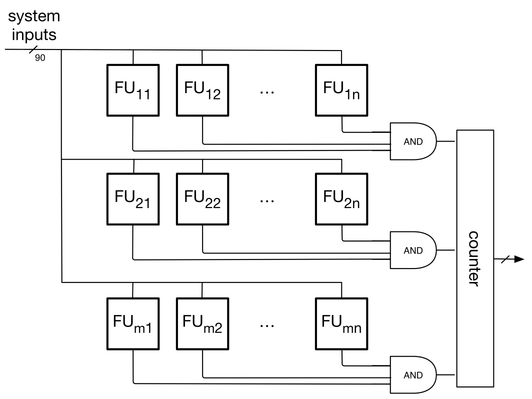
## 3 Classifier System

The system that we use as a testbench to compare the three implementation alternatives is a Classifier System [10], applied to the sonar target recognition task described in [12]. We used the sonar data set because it is a widely known and relatively difficult benchmark, while being practical to test in the hardware configuration. The focus in this paper has been to compare the efficiencies of hardware implementations, while the while suitability of the classification itself has been shown in other papers [9,16]. The system consists of $k$ Category Detection Modules (CDM), one for each category to be classified, $C_i$. The CDM with the highest output value will be detected by a maximum detector module, and the identifier of this category will be outputted, see Fig. 2(a). Each single CDM implements $m$ classification rules, and every classification rule is implemented using $n$ Functional Units (FU), one $n$-input AND gate, and one counter, see Fig. 2(b). Each FU receives all system input bits at its inputs, and produces a 1-bit output that drives one of the inputs of the AND gate. Finally, the AND gate output drives a counter that counts the number of asserted rules for the input pattern.

The FUs are the reconfigurable elements of the architecture, see Fig. 3. Each FU is formed by (1) a multiplexer which selects one field in the input data vector, a.k.a data element, $I$; (2) two functions, $f_1$ and $f_2$, that have been selected based on previous experiments, see Table 1, and whose behavior depends on the value of a constant $C$; and (3) an output multiplexer that selects which of the two functions is used in the FU. Any number and type of functions could be imagined, but in earlier works [10,12] two functions have found to be appropriate. In addition, each



(a) Classification module.



(b) Block diagram of the CDM [10].

**Fig. 2** Block diagrams of the classifier system.

| Function | Functionality |
|---|---|
| $f_1$ | $o = 1$ if $I > C$ else 0 |
| $f_2$ | $o = 1$ if $I \leq C$ else 0 |

**Table 1** Functionality of $f_1$ and $f_2$.

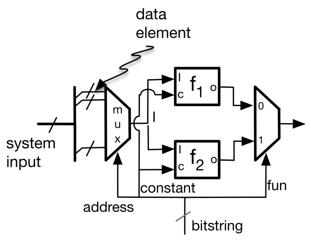


**Fig. 3** Block diagram of a functional unit.

FU is configured with a different constant value, $C$. This value, along with the data element $I$, is used by the function to compute the output.

FU behavior is managed by a configuration bitstring. The bitstring consists of three fields as shown in Fig. 4: the 6-bit field "address" selects an element in the input vector, the 1-bit field "fun" selects which of the two functions $f_1$ and $f_2$ will

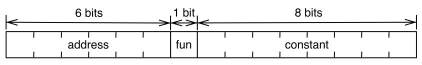


**Fig. 4** Configuration bitstring of a functional unit.

be used, see Table 1, and the 8-bit field “constant” sets the value of the constant to be used by $f_1$ and $f_2$.

The input data to be classified is simultaneously presented to each CDM using a common input bus. So, all the bits of the input data are presented to all the FU although only one field of these bits, the data element, is chosen (i.e. one byte for each FU). This data element, $I$, is then fed to the two functions in each FU.

The application data set is in the CMU Neural Networks Benchmark Collection [23] and was first used by Gorman and Sejnowski in [13]. This real-world data set consists of sonar returns from underwater targets of either a metal cylinder or a similarly shaped rock. There are 208 returns which have been divided into equally sized training and test sets of 104 returns. Each return feature vector contains 60 input data elements. The resolution of the data elements has been scaled to an 8-bit binary representation, which has been demonstrated by earlier experimentation to give adequate results [10]. This gives a total of $60 \times 8 = 480$ bits to input to the system for each return. For the sonar application we use a classifier configuration of $k = 2$, $m = 8$, and $n = 6$.

Evolving the whole classification system in one run would give a very long genome resulting in slow evolution, so an incremental approach has been chosen. Each CDM is evolved separately since there is no interdependence between the different categories. This is also true for the FU rows each CDM consists of. Although the fitness function changes between the CDMs, as detailed in [10], the evolution can be performed on every single row at a time. This significantly reduces the genome size down to $15\,\frac{bits}{FU} \times 6\,\frac{FU}{row} = 90\,$bits.

## 4 Evolutionary Algorithms

In the EHW arena, the mutation-based Evolutionary Algorithms are one of the most widely used EA because experiments demonstrate good results with this approach [10]. In particular, we use a $(\mu + \lambda)$ mutation-based EA in which the current population consists of $\mu$ individuals that are used as a source of parents to produce $\lambda$ offspring, and generate a pool of $\mu + \lambda$ individuals. In this paper, survival is performed using the elitist "truncation selection" in which the fittest of the $\mu + \lambda$ individuals (those with the highest fitness) are selected to be the $\mu$ parents for the next offspring generation. We choose that the reproduction is asexual, with each offspring coming from one single parent by mutating one or more of the parent's gene values at random (uniformly) [15]. We represent genes in two ways: as binary or integer numbers. In the binary representation, the mutation operator toggles the gene's value whereas in the integer representation the mutation operator generates an integer number within the range of gene's values and performs a XOR operation with the gene.

In this paper (1) we have selected $(1 + 8)$-mutation-based EA; (2) each individual encodes the structure of a bitstring as described in Section 3, that is, the chromosome comprises 90 genes, each one encoded as a single bit; and (3) the number of genes being modified in every generation is different for each experiment.

The fitness function is calculated as follow [10]. Each row of FUs is fed with the training vectors ($v \in V$), and the fitness is based on the row's ability to give a positive (1) output for vectors $v$ belonging to its own category ($C_v = C_i$), while giving a negative (0) output for the rest ($C_v \neq C_i$). In the case of a positive output when $C_v = C_i$, the value $A$ is added to the fitness sum. When $C_v \neq C_i$ and the row gives a negative output (value 0), 1 is added to the fitness sum. The other cases do not contribute to the fitness value. The basic fitness function $F$ for a row can then be expressed in the following way, where o is the output of the FU row:

$$F = \sum_{v \in V} x_v \qquad \text{where} \qquad x_v = \begin{cases} A \cdot o \ : C_v = C_i \\ 1 - o : C_v \neq C_i \end{cases} \tag{1}$$

For the experiments, a value of $A = 0x40$ has been used.

## 5 EHW System

Our EHW system has been designed following the two approaches introduced in Section 1, and illustrated in Fig. 1. In the hardware/software approach, the system is typically implemented on a single chip and comprises the following elements:

- A processor (ARM, MicroBlaze, PowerPC, etc.) is responsible for executing the EA software. So, the EA is embedded software, typically in C, that implements all the tasks involved in an EA except the evaluation of candidate solutions. The processor executes the EA software and reconfigures the evaluation module every time a fitness value is to be computed. After EA completion it will send the best individual, i.e. the best configuration, to the target circuit.
- An evaluation module responsible for evaluating each solution that appears during the execution of the EA. This module receives the individuals in the population from the processor, and they are evaluated using the training inputs. It

acts as a coprocessor, and it is a copy of the target circuit but with modifications to allow the evaluation of the individuals proposed by the EA.

- The target circuit containing the previous best configuration. This circuit always processes the system inputs using the configuration parameters defined during the EA search. The parameters of this module are sent from the processor after completion of each execution of the EA.
- A communication bus that connects the processor with the evaluation module, and the target circuit. Typically, the processor acts as the master in the communications. Some examples are APB or AXI buses.

In the full-hardware implementation, the system comprises three modules but the evaluation module and the target circuit have the same functionality as in the previous approach. So, the differences with respect to the hardware/software approach are:

- An EA module responsible for running the EA. The EA will be implemented as a custom digital circuit that replaces the processor in the previous approach. This module is responsible for maintaining and updating the population of individuals in the EA. It will send the individuals to the evaluation module to be evaluated, and after EA completion it will send the best individual, i.e. the best configuration, to the target circuit.
- The communication interface between the EA module and the evaluation module is typically through point-to-point connections.

In particular, we have designed three different implementations of the evolvable classifier:

1. A full-hardware implementation in which both the EA and the evaluation module, that is the classifier system, have been implemented in hardware.
2. A hardware/software implementation in which the EA has been implemented in C and run on a Zynq® XC7-Z020 PS7 5.4 hard built-in dual-core ARM® Cortex™-A9 processor, while the evaluation module has been implemented in hardware as an external co-processor.
3. A hardware/software implementation in which the EA has been implemented in C and run on a MicroBlaze™ 9.3 soft-core processor, while the evaluation module has been implemented in hardware as an external co-processor.

Thus, the evaluation module is the same for all the implementations. Next, we will describe each of these implementations. We will only describe the EA and evaluation modules, excluding the target circuit, which has been described in Section 3.

## 5.1 Full-Hardware Implementation

In this approach, the EA has been implemented in hardware. Fig. 5 shows the block diagram that comprises the `EA module`, and the `evaluation module`. Both have been described in VHDL and the complete system has been implemented on the ZedBoard™ Zynq® Evaluation Kit (XC7-Z020 ELQ484-1).

The `evaluation module` has been implemented with the structure described in Section 3 but with two modifications in its interface to achieve a significant reduction in communication overheads between the `evaluation module` and the `EA module`. Hence, the communications, in both directions, only require one clock cycle with the following modifications:

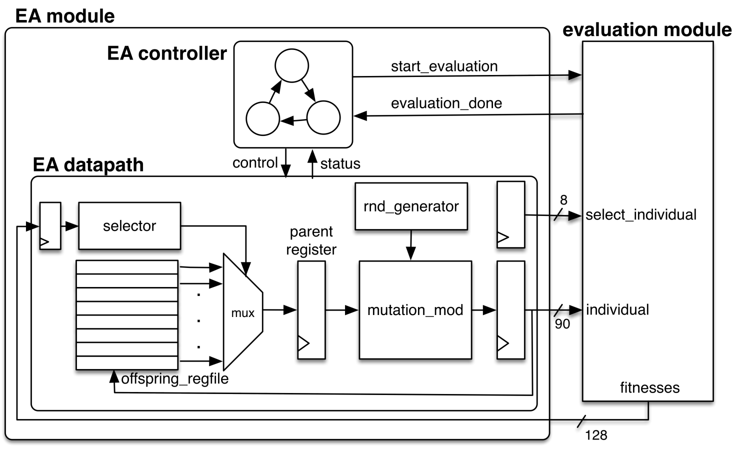


**Fig. 5** Block diagram of the full-hardware evolvable classifier. For the sake of clarity, the system inputs have not been drawn.

– Because the communications between the `EA module`, which generates the offspring, and the `evaluation module` are point to point, the width of the configuration port, `individual`, is equal to the number of bits of an individual. That is 90 bits. On the other hand, the `select_individual` port is used to indicate which of the eight individuals in the population is being written in the `individual` port.

– The `evaluation module` evaluates eight individuals simultaneously, and the eight fitness values, every fitness value being a 16-bit number, are provided simultaneously to the `EA module` using the `fitnesses` port which is $8 \times 16 = 128$ bits wide.

Fig. 5 also illustrates the block diagram of the `EA module` and its sub-modules. It has been implemented as an IP block which is fully parametrizable using three generics that define the $\lambda$ value in the mutation-based EA, $(1+\lambda)$-EA, the number of bits per chromosome, and the number of bits of the fitness value.

The `EA module` has the classical structure of an Algorithmic State Machine with a datapath, `EA datapath`, and a control unit, `EA controller`. The `EA datapath` consists of:

1. A module to generate pseudo-random numbers, `random generator`. It has been implemented according to [19]. We have designed a configurable IP block with a parametrizable number of LFSRs which are initialized by a true random number generator, TRNG, based on sampling the ring oscillator phase jitter as described in [22]. The number and width of the LFSRs are configurable at synthesis. In this paper, we have used ten 45-stage Galois LFSR with taps at positions 45, 44, 42 and 41, and cycle size $2^{45} - 1$ [36].
2. The `mutation module` that implements the mutation operator. It calculates the mutations of the parent chromosome when provided with a random number by `random generator`. This module has a fixed interface, and, in this study, it has been designed twice, with two different architectures. In the first implementation, the random generator provides one number that indicates the position of the gene to be toggled. Only one gene mutates. The experimental results, see Section 7, show that a high number of mutations provides better solutions than a low number. Therefore, we design a second `mutation module` in the full-hardware implementation to provide a greater number of mutations from parent to offspring. In order to (1) preserve the control unit of the first `mutation module`, and (2) not to modify the clock frequency, this operator should operate in just one clock cycle and with a critical path equal to or less than the one in the very simple scheme used in the first implementation. It utilizes a 90-bit vector, generated by the pseudo-random number generator, and xors it with the

parent chromosome. This mutation operator has these features: (1) the number of mutations, $n$, is random; (2) the number of mutations follows a binomial distribution, $n \sim B(90, 0.5)$, with a mean value $\langle n \rangle = 0.5 \cdot 90 = 45$ mutations [2]; and (3) this operator requires 90 XOR gates in parallel, 1 per chromosome bit, and its implementation does not change the critical path. This implementation can be scaled-up to provide any length of the random vector, either widening the number of LFSR or using the clock cycles devoted to evaluating the population to concatenate 90-bit vectors. In the case that a fixed number of mutations is required, we can split the 90-bit vector into 7-bit fields and decode them to select the genes to be toggled.

3. A register file, `offspring_regfile`, that stores the offspring. There are eight registers to store the eight individuals in the population.
4. A module, `selector`, that selects the individual with the best fitness among offspring. It receives the fitness values from `evaluation module` and selects the individual with the highest value of fitness.

The `EA controller` consists of nine states: three states for the generation of the offspring, two states for the communications with the `evaluation module`, one state to store the fitness values coming from the `evaluation module`, and two extra states to select the best offspring and become the new parent. In the simulations, the generation and sending of a new population to the `evaluation module` takes 11 clock cycles; the reading of the results and the selection of the new parent take 4 cycles.

[2] When $p = 0.5$ and $n$ is very large, usually $n \geq 30$, as in this case, the binomial distribution can be approximated by the normal distribution. So, in this case the number of mutations, $n \sim N(B \cdot p, B \cdot p \cdot q) = N(45, 27.5)$

The system has been implemented on a ZedBoard™ Zynq® Evaluation Kit (XC7-Z020 ELQ484-1) although it does not use the built-in Zynq® XC7-Z020 processor.

## 5.2 Hardware/Software Implementations

In these implementations, the EA has been implemented in software, using C language, and is responsible for maintaining and updating, generation after generation, the population in the EA. It is executed on a processor running without an operating system. On the other hand, the evaluation of the individuals is carried out in the classifier that operates as an external co-processor of the main processor. We have used two different processors: Zynq® XC7-Z020 PS7 5.4 built-in dual-core ARM® Cortex™-A9 processor system[3] and MicroBlaze™ 9.3 soft-core processor[4]. Both are defined as IP modules in the library of components that Xilinx provides for their FPGAs. Communications between the processors and the evaluation module are performed via an AXI Lite IP configured in slave mode with fourteen 32-bit registers. The global timer of the processor is used to initialize the seed of the pseudo-random number generator that provides the positions of the bits to be mutated in the parent chromosome. The registers are:

- Eight read-only (RO) registers, denoted as `R_FITNESS_<0:7>`, used by the processor to read the fitness value for each of the eight individuals in the population.

[3] 2.5 DMIPS/MHz per CPU, up to 667 MHz, with L1 and L2 caches, and single and double precision floating point units.

[4] This processor can be instantiated in any Xilinx FPGA in any number, provided that there are enough resources in the FPGA.

- Three write-only (WO) registers, R_INDIVIDUAL_LSB, R_INDIVIDUAL_MID, R_INDIVIDUAL_MSB, for sending one individual of the population to the evaluation module. The three registers total $3 \times 32 = 96$ bits. However, the individuals in the population have 90 bits, so the remaining 6 bits, located in the most significant bits of R_INDIVIDUAL_MSB, are padded with 0.
- One WO register, R_SELECT_INDIVIDUAL, to indicate which individual is being written in the registers R_INDIVIDUAL_⋆. This register receives one 1-hot byte.
- One WO control register, R_START_EVALUATION. This register has two fields: bit 0 is used to start the evaluation of the population, that is the evaluation module starts the evaluation of the offspring when this bit is asserted by the processor; bit 1 is used to reset bit 0 of the R_END_EVALUATION register.
- One RO status register, R_END_EVALUATION. This register has a single field in bit 0. This bit is asserted to 1 by the evaluation module to indicate that it has completed the evaluation of the eight individuals. This bit is set to 0 by asserting bit 1 in the R_START_EVALUATION register.

Because AXI transfers are 32 bits wide, we have been forced to redesign the interface of the classifier. Hence, the `individual` port is now 32-bit, not 90-bit as in the hardware implementation, and the writing of a new configuration requires three write operations of the processor into R_INDIVIDUAL_⋆ registers plus one additional writing of the R_SELECT_INDIVIDUAL register that stores the value of the `select_individual` port. Likewise, the reading of the fitness values requires eight read operations, one per R_FITNESS_<0:7> register.

Alg. 1 presents the pseudo-code of the C program. The individual's chromosome is represented as an array of 18 genes ($3\ \frac{genes}{row} \times 6\ \frac{FU}{row}$), each gene being represented by a pointer to integer, and the population being represented as a pointer to an array of such data. The range of values that any gene can take depends on the field of the bitstring represented by the specific gene: integers representing the "address" field can take values in the range [0, 63], those representing the "fun" field can take values in the range [0, 1], and so on. These data structures have been chosen to speed up the software performance as much as possible. The software is responsible for:

1. Generating offspring (lines 3 to 7). The evaluation module operates on a binary representation of the chromosomes but, as stated above, the software operates on an integer representation. Line 7 translates from 18-integer representation to 90-bit binary representation.
2. Sending them to the co-processor (lines 8 to 11).
3. Waiting for the classifier to end the evaluation of the $\lambda$ individuals in the population by actively polling register `R_END_EVALUATION` (lines 12 to 13).
4. Reading the fitness values and selecting the best offspring, based on the fitness calculated by the classifier, to become the progenitor of the next generation (lines 15 to 18).

The software has been compiled using the options -O2 -g3 and run in bare metal mode on the chosen processor.

**Algorithm 1:** Pseudo-code of the software implementations.

```
while generation < maxGens do
    for i = 1; $i < \lambda$; i++ do
        if i==1 then
            offspring(i) ← parent;
        else
            offspring(i) ← mutate(parent);
        offspring_bin ← int2bin(offspring(i));
        writeIP(R_INDIVIDUAL_LSB, offspring_bin [31:0]);
        writeIP(R_INDIVIDUAL_MID, offspring_bin [63:32]);
        writeIP(R_INDIVIDUAL_MSB, offspring_bin [89:64]);
        writeIP(R_SELECT_INDIVIDUAL, one-hot(i));
    writeIP (R_START_EVALUATION, 0x1);
    while readIP(R_END_EVALUATION) ≠ 0x1 do
        –
    writeIP (R_START_EVALUATION, 0x20);
    initialize(best_fitness);
    for i = 1; $i < \lambda$; i++ do
        new_fitness ← readIP(R_FITNESS_<i>) ;
        if new_fitness ≥ best_fitness then
            parent ← offspring(i);
            best_fitness ← new_fitness
```

## 6 Methodology

We have implemented the three designs presented in Section 5 on a ZedBoard™ Zynq® Evaluation Kit (XC7-Z020 ELQ484-1) using Xilinx Vivado® Design Suite 2014.2. In the comparison, we have avoided introducing any bias towards any of the alternatives. For example, the full-hardware implementation could work at 388

MHz if the EA module had been pipelined, or even at higher frequencies using deep pipelining in the EA and Evaluation modules. Similarly, its power consumption could be reduced using clock-gating techniques. These techniques would have biased the results for the full-hardware implementation.

We have evaluated the implementations for a set of different configurations of the EA. Each one is defined by (1) the implementation approach, either full-hardware or hardware/software, and (2) the mutation operator. We have denoted each configuration with a different tag which contains four characters: the first character can be either h or s, and indicates whether the configuration has been implemented in full-hardware or hardware/software, respectively; the second character is the number of mutations, $N$; the third can be either r or f, and indicates whether the number of mutations has been chosen randomly in the range $[1, N]$ or is fixed at $N$; finally, the fourth character can be either b or i and indicates whether the mutations are performed on the binary or integer representations of the chromosome. For example, h1fb is the tag for the hardware implementation of an EA where only one bit of the parent chromosome changes to generate a new offspring. Similarly, s4ri denotes the EA in which the mutation operator mutates a random number of integers in the range $[1, 4]$. For this case, the chromosome is formed by 18 genes, and each one is represented by an integer in the hardware/software implementation that is compacted to a 90-bit number when it is sent to the classifier.

We have measured the following metrics for each of the three designs:

– The size has been measured as the FPGA slices required by each implementation. These comprise the resources used by the custom-designed logic, the memories and the processor in the design.

- The timing has been measured, using static timing analysis, as the maximum clock frequency the design can work. The designs have only one clock, and we define its frequency in the User Constraint File. Primary-input-to-register and register-to-primary-output delays have been constrained so that input and output paths have to fit into a half period. No combinational paths exist from primary inputs to primary outputs. All paths have positive slack after timing closure. Nevertheless, in this paper, the training vectors are provided within the FPGA so that there are not any timing-critical input or output ports. In those designs that contain a processor, the frequency of the EHW is defined by the maximum speed of the AXI Lite interface.
- The power consumption has been estimated using the built-in power estimator in Vivado 2014.2 Suite, assuming typical process and working conditions, and a toggle rate of 0.125 with 50% of the time high for the logic.
- The speed of operation has been measured as the execution time of the actual circuit on the ZedBoard. The time has been measured using the global timer of the processor for the hardware/software implementation and a hardware counter for the hardware implementation. Every configuration has been run 20 times. We average the execution time for each configurationon task and perform hypothesis test. First, we verify whether the 20 measurements follow a normal distribution using the Lilliefors test. If so then we run the two-sample t-test for every pair of configurations with the null hypothesis being that both samples come from independent random samples from normal distributions with equal means. If the null hypothesis is rejected, then the samples do not come from distributions with equal mean. Additionally, we have estimated the confidence interval for all the pairs and ranked the results. If the Lilliefors test rejects the

sample normality, then we run a one-way balanced ANOVA test to determine if the differences between samples are statistically significant.

Area, timing and power-consumption have been measured by comparing two EA configurations: h1fb vs. s1fb. The speed of operation has been measured for up to fifteen EA configurations.

## 7 Results

Table 2 presents the use of FPGA resources, and the timing for the three implementations for the implementation of h1fb and s1fb EA configurations. The full-hardware implementation works at 116 MHz. The Zynq implementation has been synthesized with a 667 MHz processor clock frequency, 500 MHz DDR clock frequency, and 100 MHz for the Programmable Logic (PL). The clock frequency for the PL in the Zynq implementation is constrained by the nearest operating frequency of the AXI Lite interface below 115 MHz. The AXI Lite interface is used to connect the processor with the evaluation module. Finally, the MicroBlaze implementation has been synthesized at 100 MHz, the maximum clock frequency, and with support for Debug + UART. There are two reasons to include Debug+UART support. On the one hand, these two components are required for communication with the external PC that monitors and logs the evolution. On the other hand, they do not slow down the processor speed.

The MicroBlaze implementation has 256 KB data and instruction memories because the C elf file generated by the compiler does not fit into smaller memories. The `EA module` module accounts for 2206 slices and most of them are devoted

| Feature | Hw | Sw-Zynq | Sw-Blaze |
|---|---|---|---|
| Total Slices PL | 3715 | 3669 $^{(*)}$ | 5412 |
| Slice LUTs | 1134 | 1229 $^{(*)}$ | 2132 |
| Slice Regs | 2467 | 1767 $^{(*)}$ | 2575 |
| BRAM Tile | 17 | 17 $^{(*)}$ | 81 |
| Clock Processor (MHz) | — | 667 | 100 |
| Clock PL (MHz) | 116 | 100 | 100 |
| $\mu$P | No | Yes | Yes |

**Table 2** Resource utilization comparison. $^{(*)}$ These figures do not account for the resources needed to implement the Zynq® XC7-Z020.

to storing the population of eight 90-bit individuals. Again, AXI Lite interface has been used to connect the processor with the evaluation module.

The full-hardware implementation requires the fewest resources in the FPGA: 101.25% of the total number of slices of the Zynq implementation, and the same number of BRAMs in the PL. However, the figures for the Zynq implementation do not take into account the resources required for the processor[5], such as all the memories required to implement L1 and L2 caches. Regarding the comparison with MicroBlaze implementation, the full-hardware implementation requires 68.64% of the total number of slices and 20.99% of the BRAMs. For the MicroBlaze processor, data and instruction memories are implemented in the PL, and they account for the high number of BRAMs.

Regarding power-consumption, full-hardware consumes 290 mW, MicroBlaze consumes 490 mW, and Zynq between 1.233 W and 1.463 W. So, full-hardware is the lowest power-consumption design: it consumes 72.5% of the power of the MicroBlaze implementation –mainly because of the number of memories of the

[5] These figures are not available in Vivado 2014.2.

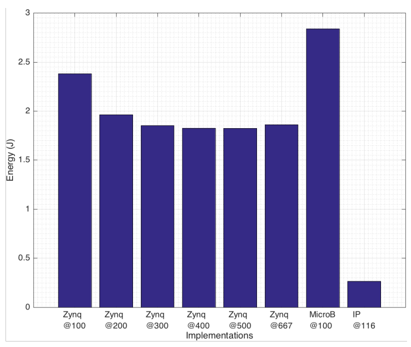


**Fig. 6** Energy consumption of the different implementations. X axis labels represents the implementation followed by the processor clock frequency, and y-axis is the energy consumption in joules.

MicroBlaze–, and between 19.82% and 24.62% of the power consumption of the Zynq.

Fig. 6 shows the comparison of the three implementations in terms of their energy consumption to run 16K generations using h1fb and s1fb configurations. The full-hardware implementation consumes the least energy, 0.226 J, and the MicroBlaze the most, 2.839 J.

Fig. 7 presents the comparison between power-consumption and the averaged execution time over 20 runs, and 16K generations each run, for the h1fb and s1fb configurations. The number of generations in these experiments is fixed, so the execution time is practically the same in the 20 runs of each configuration. The full-hardware implementation has been executed using a 116 MHz clock, the MicroBlaze implementation using a 100 MHz clock for the processor and 100 MHz clock for the PL. The Zynq implementation has been studied for a 100 MHz PL clock, and different clock frequencies for the processor and DDR. Hence, the dots in the right lower corner have been measured for 500 MHz DDR and processor frequencies of

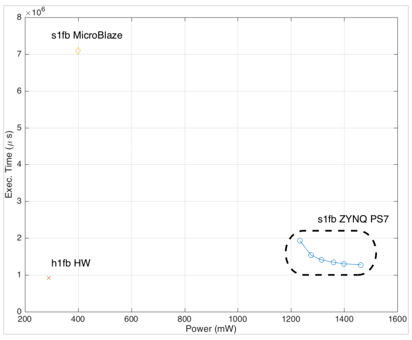


**Fig. 7** Execution time vs Power consumption.

100 (the lowest power consumption), 200, 300, 400, 500, and 667 MHz (the highest power consumption).

The full-hardware implementation is the fastest: between $\times 1.39$ and $\times 2.11$ faster than Zynq, and $\times 7.74$ faster than MicroBlaze. There are two reasons: firstly, it runs at a higher clock frequency; secondly, it requires a lower number of clock cycles than the hardware/software implementations to run the EA and to send and receive the individuals from the evaluation module. The hardware's inherent parallelism allows implementing EA operators that perform operations in just one cycle while the same process requires many clock cycles to be executed by the processors. Similarly, Zynq implementation performs better than MicroBlazeas much in performance as in energy consumption. For this reason, from now on, we focus our analysis of the performance on the two best implementations.

Next, we measure two different execution times of the two implementations: (1) the execution time to run 16K generations, and (2) the execution time to find a solution within a 2.8% of the maximum fitness. The rationale of the second type of experiments is that those mutation operators implemented in software having a more complex mechanism may find better solutions in a shorter time than the full-hardware implementations. This approach to select the best solutions is widespread in learning or adaptive real-time systems [26,31]. In these experiments, the maximum number of generations is 48K. The hardware/software implementation is configured with 667 MHz and 500 MHz clocks for the processor and DDR, respectively.

For these experiments, we compare fifteen EA configurations. Table 3 presents the ranking of the configurations according to the mean execution time to run 16K generations. The Lilliefors test indicates that the 20 samples of each configuration follow a normal distribution and the confidence intervals reject the null hypothesis in all except one case: the two full-hardware implementations that have the same timing, as detailed in Section 5.1. For all the other experiments the differences are statistically significant. The full-hardware implementations give the best results. In the hardware/software configurations, the execution time increases with the number of mutations, either per bit or gene, and similarly for the experiments with a random number of mutations. These results are due mainly to the fact that the number of clock cycles required by the hardware approach is much smaller (6521 clock cycles per generation) than the number of clock cycles required by the software approaches. If eHW had been designed to work at high speed, then another source of variation would be the different clock frequencies for the full-hardware and software implementations, because in software the operating clock frequency is limited by the AXI allowed frequencies.

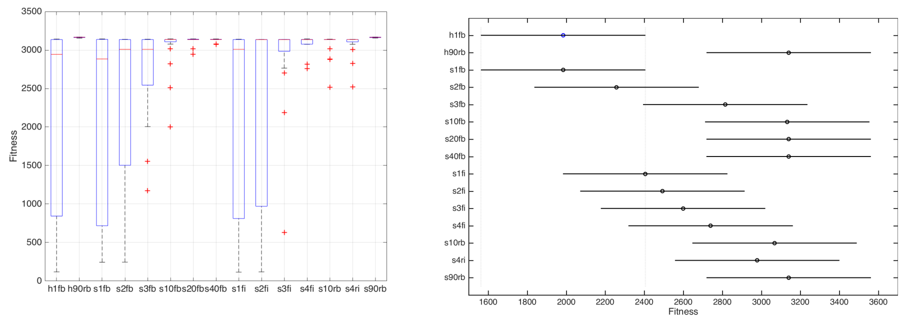
Fig. 8(a) shows the boxplot for the values of fitness for the best individual in the last generation – that is, the best individual so far – for 20 runs in each experiment. The Lilliefors test indicates the best individual fitnesses do not follow a normal distribution. Fig. 8(b) presents the results of the one-way balanced ANOVA. This

| Rank | Configuration | mean (s) | ci (s) |
|---|---|---|---|
| 1 | h1fb | 0.917 | [0 , 0] |
| 2 | h90rb | 0.917 | [-0.3547, -0.3543] |
| 3 | s1fb | 1.271 | [-0.0009, -0.0005] |
| 4 | s1fi | 1.273 | [-0.0178, -0.0174] |
| 5 | s2fb | 1.290 | [-0.0149, -0.0146] |
| 6 | s2fi | 1.305 | [-0.0109, -0.0106] |
| 7 | s3fb | 1.316 | [-0.0158, -0.0154] |
| 8 | s4ri | 1.331 | [-0.0063, -0.0059] |
| 9 | s3fi | 1.338 | [-0.0343, -0.0340] |
| 10 | s4fi | 1.372 | [-0.0705, -0.0701] |
| 11 | s10rb | 1.408 | [-0.0842, -0.0836] |
| 12 | s10fb | 1.492 | [-0.2485, -0.2473] |
| 13 | s20fb | 1.740 | [-0.1907, -0.1879] |
| 14 | s90rb | 1.929 | [-0.3137, -0.3109] |
| 15 | s40fb | 2.241 | |

**Table 3** Mean execution time and confidence intervals after running 16K generations for all configurations. Each configuration has been run 20 times. mean stands for mean execution time and ci for confidence interval, using the two sample t-test, of the difference between a configuration and the next ranked configuration.

test determines that the differences are statistically significant ($p < 0.05$) between those configurations that their intervals are disjoint.

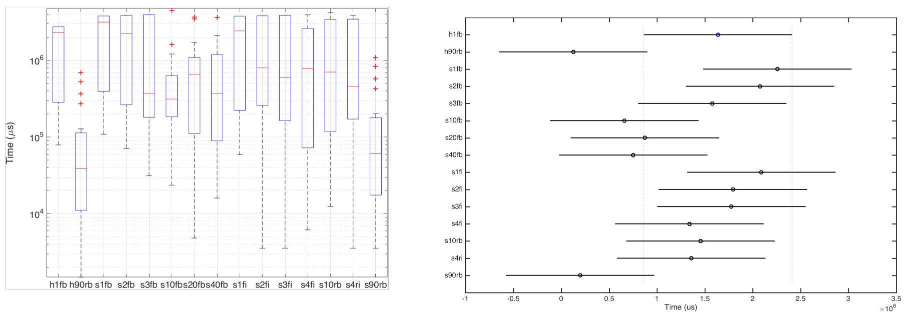
The two EA configurations that mutate a single bit, h1fb and s1fb, produce the worst results, and the fitness of the best solutions improves as the number of mutations increases. However, the fact that both configurations perform so badly suggests that the problem is not inherent to the hardware implementation, but to the mutation operator chosen.



(a) Boxplot (b) ANOVA

**Fig. 8** (a) Boxplot graph of the fitness of the best individual after 16K generations for the training set. Each experiment has been run 20 times. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually. (b) ANOVA analysis. Each group mean is represented by a dot, and the interval is represented by a line extending out from the symbol. Two group means are significantly different if their intervals are disjoint.

Fig. 9 shows the execution time to find the solutions within 2.8% of the maximum fitness. Fig. 9(a) shows the boxplot. The Lilliefors test indicates that each configuration samples do not follow a normal distribution. The ANOVA test, in Fig 9(b), determines the results of the configurations are statistically different ($p < 0.05$) if the intervals do not overlap. The mutation operators that mutate a higher number of bits produce the best solutions in a lower number of generations, and this happens both in the hardware and software implementations. In addition, the mutation operator h90rb requires the same number of cycles per generation as h1fb, since both operators require the same number of cycles to generate a new offspring, but h90rb finds the solutions in much less time than h1fb. Again, these results are because (1) the hardware approaches, h1fb and h90rb, work at a higher clock frequency than the Zynq® XC7-Z020, (2) the number of clock cycles that is required for



(a) Boxplot (b) ANOVA

**Fig. 9** (a) Boxplot graph of the execution time to find the best solutions for both kinds of implementations and different EAs. Each experiment has been run 20 times. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually. (b) ANOVA analysis. Each group mean is represented by a dot, and the interval is represented by a line extending out from the symbol. Two group means are significantly different if their intervals are disjoint.

the hardware approach is much smaller than for the software approaches, and (3) the number of generations of the new hardware approach is as low as the best software approaches. The number of mutations in the configurations s90rb and h90rb follows a normal distribution with average value 45. So, in average, the effective mutation rate of these two configurations is 0.5. Keeping the number of genes that are modified high has been proved to be beneficial in this kind of problems characterized by a strong epistasis in which improvements in the fitness require simultaneous mutations in multiple genes. Additionally, the elitist truncation selection, that preserves the best solutions found so far, is a high-pressure selection mechanism, and the use of an aggressive mutation rate balances such a strong selection operator. Finally, this result is in line with previous results in the literature [27].

Finally, we are going to estimate the performance of both types of configuration in the case that the number of vectors or generations were several orders of magnitude greater. A single generation is the execution of the EA, that comprises the communication between EA and Evaluation modules, followed by the execution of the evaluation module. Eq. (2) estimates the execution time, $L$, for $g$ generations where $t_{ea}$ is the execution time of the EA, and $t_{ev}$ is the execution time of the evaluation. $t_{ea} = 0.1292\ \mu s$ for the full-hardware implementation and $t_{ea} = 13.1771\ \mu s$ for the Zynq implementation at 667 MHz. In both implementations $t_{ev} = \frac{\text{cycles}}{f_{clk}}$ where cycles is the number of clock cycles to complete the evaluation of 8 individual, $f_{clk} = 116$ MHz for the full-hardware implementation and $f_{clk} = 100$ Mhz for the hardware/software implementation. Eq. (3) defines the speedup, S, of execution time, where $L_{Hw}$ and $L_{Zynq}$ are the execution times for the full-hardware and Zynq implementations, respectively. Fig. 10 plots Eq. (3) for a different number of cycles in the evaluation phase. The number of cycles is related to the number of training vectors, so this figure illustrates how the speedup changes with a higher number of training vectors. According to this graph, the full-hardware implementation is $\times 1.17$ faster than Zynq implementation even when the number of training vector increases $\times 100$.

$$L = g \cdot (t_{ea} + t_{ev}) \tag{2}$$

$$S = \frac{L_{Hw}}{L_{Zynq}} \tag{3}$$

Applications with more generations will not change the comparisons because $S$ does not depend on $g$. Regarding the mutation, the implementation of a more complex operator working at 116 MHz will operate using the same number of clock

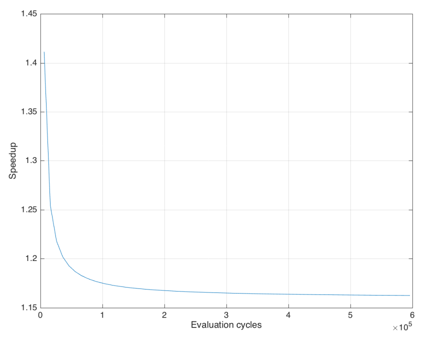


**Fig. 10** Speedup full-hardware vs Zynq implementations for a different number of cycles in the execution of the evaluation.

cycles because this operator is not in the critical path of the design. So, this will not change the results significantly. For much more complex operators, the EA module can be deep-pipelined at much higher clock frequency.

To summarize, in all the metrics related to the characteristics and performance of a digital circuit, the full-hardware implementation equals or overcomes the hardware/software implementations.

## 8 Conclusions

In this paper, we present three implementations of an online EHW and compare their features using the most relevant metrics in the design of hardware: area, timing, power consumption, energy consumption, and performance.

To our best knowledge, there is no earlier paper devoted to implementing and comparing different implementations of an EHW system on a current 28 nm process technology FPGA using two different modern processors, Zynq® XC7-Z020 PS7 5.4 built-in dual-core ARM® Cortex™-A9 and MicroBlaze™ 9.3 soft-core processors, and one full-hardware implementation. These comparisons illustrate the expected achievements of the different alternatives using modern IP processors on current FPGA technologies.

The full-hardware implementation requires the fewest resources in the FPGA: 101.25% of the total number of slices of the Zynq implementation, and the same number of BRAMs in the PL. However, the figures for the Zynq implementation do not take into account the resources required for the processor. Regarding the comparison with MicroBlaze implementation, the full-hardware implementation requires 68.64% of the total number of slices and 20.99% of the BRAMs.

Regarding power consumption, the full-hardware implementation consumes $\times 0.725$ of the power of the MicroBlaze implementation and $\times 0.198$ of the power of the Zynq implementation. The power consumption of the hardware/software implementations is high because of the presence of the processors.

The energy consumption of the full-hardware implementation is the lowest as a result of its high performance and low power consumption when compared to hardware/software implementations. Similarly, the full-hardware implementation requires the lowest number of resources in the FPGA because of the lack of a processor.

The full-hardware implementation gives the highest performance: up to $\times 7.74$ faster than the hardware/software implementation with the worst performance, and $\times 1.39$ faster than the hardware/software implementation using embedded processors running at 667 MHz. There are two reasons. First, it runs at a higher clock frequency: 116 MH vs. 100 MHz of the hardware/software implementation. Second, it requires a lower number of clock cycles than the hardware/software implementations (1) to run the EA and (2) to send and receive the individuals from the evaluation module. The hardware's inherent parallelism allows it to implement EA operators that perform operations in just one clock cycle, while the same operation requires many clock cycles to be executed by the processor.

Therefore, the overall achievement of the full-hardware implementation is better, provided that the evolutionary operators do not have a convoluted hardware implementation that impacts in the circuit timing. However, the hardware/software implementations are more flexible because of its ease to test different implementation alternatives before choosing the final version.

As a result of the previous comparisons, the hardware/software implementation is recommended, principally:

1. When flexibility is mandatory in the evolutionary algorithm. Full-hardware implementation cannot be modified easily once the system has been designed and deployed.
2. In non-high-performance applications (using Zynq® XC7-Z020), regardless of the power consumption and resource utilization.
3. For low-power consumption, regardless of the performance and the resources.

In any other cases,the full-hardware implementation is superior. That is:

1. Very high-frequency EHW, such as deep-pipelined circuits. Software solutions cannot take advantage of the performance of these designs because of the limitations of the AXI interface that cannot work at frequencies higher than 250 MHz.
2. Systems with low power consumption and high-performance. Full-hardware implementation is the only alternative to achieve both goals simultaneously.
3. Complex systems with a high gate count, so there are tight constraints on resource utilization of the subsystems, which simultaneously require high perfor-

mance. Again, full-hardware implementation is the only alternative to achieve both goals simultaneously.

We have established the previous conclusions using an application with features similar to those of the most typical applications in current EHW. Nowadays, new applications based on bio-inspired systems implemented on FPGA are being presented, such as deep machine learning. The comparison of the five metrics for this class of systems will require of new experiments using entirely different implementations from those presented in this work, with new types of processors located outside the FPGA and different communication architectures. We will devote future works to studying these applications.

**Acknowledgements** This work is supported by the Spanish Research Grant TIN2015-65460-C2, and NILS Grant ABEL-IM-2014A.

## References

1. Balleri, A.: Biologically inspired radar and sonar target classification. Ph.D. thesis, UCL (University College London) (2010)
2. Blodget, B., James-Roxby, P., Keller, E., McMillan, S., Sundararajan, P.: A Self-reconfiguring Platform, pp. 565–574. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). DOI 10.1007/978-3-540-45234-8_55. URL http://dx.doi.org/10.1007/978-3-540-45234-8_55
3. Brighton, H., Mellish, C.: Advances in instance selection for instance-based learning algorithms. Data mining and knowledge discovery **6**(2), 153–172 (2002)
4. Cancare, F., Bartolini, D.B., Carminati, M., Sciuto, D., Santambrogio, M.D.: On the evolution of hardware circuits via reconfigurable architectures. ACM Trans. Reconfigurable Technol. Syst. **5**(4), 22:1–22:22 (2012). DOI 10.1145/2392616.2392620. URL http://doi.acm.org/10.1145/2392616.2392620

5. Cancare, F., Santambrogio, M.D., Sciuto, D.: A direct bitstream manipulation approach for virtex4-based evolvable systems. In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems, pp. 853–856 (2010). DOI 10.1109/ISCAS.2010.5537429

6. Dobai, R., Sekanina, L.: Towards evolvable systems based on the xilinx zynq platform. In: Evolvable Systems (ICES), 2013 IEEE International Conference on, pp. 89–95 (2013). DOI 10.1109/ICES.2013.6613287

7. Dobai, R., Sekanina, L.: Low-level flexible architecture with hybrid reconfiguration for evolvable hardware. ACM Trans. Reconfigurable Technol. Syst. **8**(3), 20:1–20:24 (2015). DOI 10.1145/2700414. URL `http://doi.acm.org/10.1145/2700414`

8. Glette, K.: Design and implementation of scalable online evolvable hardware pattern recognition systems. Ph.D. thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, The address of the publisher (2008). An optional note

9. Glette, K., Kaufmann, P., Assad, C., Wolf, M.T.: Investigating evolvable hardware classification for the biosleeve electromyographic interface. In: 2013 IEEE International Conference on Evolvable Systems (ICES), pp. 73–80. IEEE (2013). DOI 10.1109/ICES.2013.6613285. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6613285`

10. Glette, K., Torresen, J., Kaufmann, P., Platzner, M.: A comparison of evolvable hardware architectures for classification tasks. In: Evolvable Systems: From Biology to Hardware, *Lecture Notes in Computer Science*, vol. 5216, pp. 22–33. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-85857-7_3. URL `http://link.springer.com/10.1007/978-3-540-85857-7_3`

11. Glette, K., Torresen, J., Kaufmann, P., Platzner, M.: A Comparison of Evolvable Hardware Architectures for Classification Tasks, pp. 22–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-85857-7_3. URL `http://dx.doi.org/10.1007/978-3-540-85857-7_3`

12. Glette, K., Torresen, J., Yasunaga, M.: An online ehw pattern recognition system applied to sonar spectrum classification. In: Proceedings of the 7th International Conference on Evolvable Systems: From Biology to Hardware, *Lecture Notes in Computer Science*, vol. 4684, pp. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). DOI 10.1007/978-3-540-74626-3_1. URL `http://link.springer.com/10.1007/978-3-540-74626-3_1`

13. Gorman, R.P., Sejnowski, T.J.: Analysis of hidden units in a layered network trained to classify sonar targets. Neural Networks **1**(1), 75–89 (1988). DOI http://dx.doi.org/10.1016/0893-6080(88)90023-8. URL `http://www.sciencedirect.com/science/article/pii/0893608088900238`

14. Greenwood, G.W., Tyrrell, A.M.: Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems. IEEE Press Series on Computational Intelligence. John Wiley & Sons, Inc (2006)

15. Jong, K.A.D.: Evolutionary Computation: A Unified Approach. MIT Press (2006). URL `https://books.google.co.in/books?id=OIRQAAAAMAAJ`

16. Kaufmann, P., Glette, K., Gruber, T., Platzner, M., Torresen, J., Sick, B.: Classification of electromyographic signals: Comparing evolvable hardware to conventional classifiers. IEEE Transactions on Evolutionary Computation **17**(1), 46–63 (2013). DOI 10.1109/TEVC.2012.2185845

17. Lee, S.Y., Hong, J.H., Hsieh, C.H., Liang, M.C., Chien, S.Y.C., Lin, K.H.: Low-power wireless ECG acquisition and classification system for body sensor networks. IEEE Journal of Biomedical and Health Informatics **19**(1), 236–246 (2015). DOI 10.1109/JBHI.2014.2310354

18. López, B., Valverde, J., de la Torre, E., Riesgo, T.: Power-aware multi-objective evolvable hardware system on an fpga. In: Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on, pp. 61–68 (2014). DOI 10.1109/AHS.2014.6880159

19. Martin, P.: An analysis of random number generators for a hardware implementation of genetic programming using FPGAs and Handel-C. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '02, pp. 837–844. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002). URL `http://dl.acm.org/citation.cfm?id=646205.682460`

20. Mora, J., Otero, A., de la Torre, E., Riesgo, T.: Fast and compact evolvable systolic arrays on dynamically reconfigurable fpgas. In: Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on, pp. 1–7 (2015). DOI 10.1109/ReCoSoC.2015.7238087

21. Salvador, R.: Evolvable hardware in fpgas: Embedded tutorial. In: 2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), pp. 1–6 (2016). DOI 10.1109/DTIS.2016.7483877

22. Schellekens, D., Preneel, B., Verbauwhede, I.: FPGA vendor agnostic true random number generator. In: Field Programmable Logic and Applications, 2006. FPL '06. International Conference on, pp. 1–6 (2006). DOI 10.1109/FPL.2006.311206
23. Sejnowski, T., Gorman, R.P.: CMU neural networks benchmark collection. `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/` (1995)
24. Sekanina, L.: Image filter design with evolvable hardware. In: Proceedings on Applications of Evolutionary Computing, Lecture Notes in Computer Science, pp. 255–266. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). DOI 10.1007/3-540-46004-7_26. URL `http://dx.doi.org/10.1007/3-540-46004-7_26`
25. Sekanina, L.: Evolvable Components - From Theory to Hardware Implementations. Natural Computing Series. Springer Verlag (2003)
26. Shi, Y., Eberhart, R.C.: Fuzzy adaptive particle swarm optimization. In: Proceedings of the 2001 Congress on Evolutionary Computation, vol. 1, pp. 101–106 (2001). DOI 10.1109/CEC.2001.934377. URL `http://dx.doi.org/10.1109/CEC.2001.934377`
27. Stomeo, E., Kalganova, T., Lambert, C.: Chose the right mutation rate for better evolve combinational logic circuits. International Journal of Computer Intelligence **2**(2), 277–286 (2006)
28. Torresen, J.: Evolvable hardware-a short introduction. Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems **1**, 674–677 (1997)
29. Torresen, J.: Incremental evolution of a signal classification hardware architecture for prosthetic hand control. KES Journal **12**(3), 187–199 (2008). URL `http://content.iospress.com/articles/international-journal-of-knowledge-based-and-intelligent-engineering-systems/kes00160`
30. Torresen, J., Senland, G.A., Glette, K.: Partial reconfiguration applied in an on-line evolvable pattern recognition system. In: NORCHIP, 2008., pp. 61–64 (2008)
31. Urbanowicz, R.J., Moore, J.H.: Learning classifier systems: A complete introduction, review, and roadmap. J. Artif. Evol. App. **2009**, 1:1–1:25 (2009). DOI 10.1155/2009/736398. URL `http://dx.doi.org/10.1155/2009/736398`
32. Vašíček, Z., Sekanina, L.: Hardware accelerator of cartesian genetic programming with multiple fitness units. Computing and Informatics **29**(6), 1359–1371 (2010)

33. Vašíček, Z., Sekanina, L.: Evolutionary approach to approximate digital circuits design. IEEE Trans. Evol. Comput. **19**(3), 432–444 (2015). DOI 10.1109/TEVC.2014.2336175. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6848841`

34. Vašíček, Z., Žádník, M., Sekanina, L., Tobola, J.: On Evolutionary Synthesis of Linear Transforms in FPGA, pp. 141–152. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-85857-7_13. URL `http://dx.doi.org/10.1007/978-3-540-85857-7_13`

35. Wang, J., Chen, Q.S., Lee, C.H.: Design and implementation of a virtual reconfigurable architecture for different applications of intrinsic evolvable hardware. IET Computers Digital Techniques **2**(5), 386–400 (2008). DOI 10.1049/iet-cdt:20070124

36. Ward, R., Molteno, T.: Table of linear feedback shift registers. Tech. rep., Department of Physics, University of Otago, Box 56, Dunedin, New Zealand (2007)

37. Yao, X., Higuchi, T.: Promises and challenges of evolvable hardware. IEEE Trans. Syst., Man, Cybern. C **29**(1), 87–97 (1999). DOI http://dx.doi.org/10.1109/5326.740672. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=740672`