Published in IET Computers & Digital Techniques
Received on 21st September 2012
Revised on 11th May 2013
Accepted on 13th May 2013
doi: 10.1049/iet-cdt.2012.0109

ISSN 1751-8601

Field programmable gate arrays-based differential evolution coprocessor: a case study of spectrum allocation in cognitive radio network

Kiran Kumar Anumandla[1], Rangababu Peesapati[1], Samrat L. Sabat[1], Siba K. Udgata[2], Ajith Abraham[3]
[1]*School of Physics, University of Hyderabad, Hyderabad 500046, India*
[2]*School of Computer and Information Sciences, University of Hyderabad, Hyderabad 500046, India*
[3]*Machine Intelligence Research Labs, Scientific Network for Innovation and Research Excellence P.O. Box 2259, Auburn Washington 98071-2259, USA*
E-mail: slssp@uohyd.ernet.in

Abstract: In this study, a scalable coprocessor for accelerating the Differential Evolution (DE) algorithm is presented. The coprocessor is interfaced with PowerPC embedded processor of Xilinx Virtex-5 FX70T Field Programmable Gate Array. In the proposed design, the DE algorithm module is tightly coupled with fitness function module to reduce communication and control overhead. The fixed point DE algorithm is implemented in the coprocessor whereas both fixed and floating point DE are implemented in the embedded processor. Performance of the coprocessor is evaluated by optimising benchmark functions of different complexities. The implementation results show that the coprocessor is 73.14–160.2× and 2.19–27.63× faster compared to the software execution time of the floating and fixed point algorithm respectively. As a case study, spectrum allocation problem of cognitive radio network is evaluated with the coprocessor. Results show an acceleration of 76.79–105× and 5.19–6.91× with respect to floating and fixed point DE in embedded processor. It is also observed that the application occupies 56% of BRAM, 54% of DSP48E, 16% of slice LUTs and maximum frequency of operation as 63.55 MHz in a Virtex-5 FPGA. This type of coprocessor is suitable for embedded applications where the fitness function remains unchanged.

1 Introduction

Differential evolution (DE) algorithm is an evolutionary computation method and has been applied in diverse domains of science and engineering applications [1]. DE finds optimal values for a set of parameters by making repeatedly pseudo-random changes to their values. The number of parameters are referred as dimension of the problem. After making changes, the algorithm evaluates the fitness of the solution. DE algorithm became a popular evolutionary algorithm because (i) it is simple to implement, (ii) it has better performance in comparison with other evolutionary algorithms (EA) and (iii) it has less number of control parameters and less space complexity [2]. Most of the evolutionary techniques have been implemented in a high end desktop computer/processors to solve optimisation problems. The applications like motion estimation [3], pole-placement design of infinite-impulse-response filter [4], future generation evolvable machines [5] use evolutionary algorithm to derive optimal solutions. These applications generally uses low-performance microprocessors with limited computational resources, rather than high-performance desktop personal computers/processors to execute the evolutionary algorithms. The time consuming evolution process limits the use of evolutionary algorithms in embedded applications. This leads to slow execution speed of the algorithms in embedded processor. In order to meet the real time execution speed requirement, one can either proceed with the parallelisation of the algorithm or implement the design onto the hardware. There are several hardware platforms such as microcontrollers (μC), digital signal processors (DSP), field programmable gate arrays (FPGA) and application specific integrated circuits (ASIC) are used for developing an embedded system. Platforms like μC and DSP are revolving around firmware development using software methodologies rather than development of hardware for the application [6]. FPGA development platforms support both hardware-based approach (system developed entirely in the hardware) and processor-based approach (system developed entirely in the firmware). It has the flexibility to customise the hardware design by adding any combination of peripherals and controllers which are not available in microcontroller or DSP processor-based system. Owing to the above reasons, recently evolutionary algorithms like particle swarm optimisation (PSO), genetic algorithm have been implemented in the FPGA [7, 8]. The execution time of the DE algorithm increases with the increase in complexity of the function to be optimised. Owing to this, DE algorithm is not suitable for implementation in low end processors for real-time/online applications involving complex optimisation. Thus there is an increasing demand to define an architecture and implement the algorithm in the FPGA to meet the real-time execution speed requirement. DE algorithm can be implemented in both embedded processor and hardware using either fixed point or floating point arithmetic. Although floating point DE will give better accuracy but at the expense of high computation cost. In embedded processor, floating point DE reduces the execution speed approximately by 5–40×.

The objective of this work is to implement the DE algorithm in the FPGA platform to accelerate the optimisation speed. This paper focuses on only improving the speed of the optimisation time and not to improve the quality of solutions. For improving the quality of solution different variants of DE algorithm can be explored. The DE algorithm has three major computational operations (i) random number generation (RNG), (ii) objective function evaluation and (iii) updating the solution. For optimising a simple testbench function (Rosenbrock), the profiling result of DE algorithm shows that RNG and the optimisation algorithm (except the fitness function evaluation) consumes 90% of the total execution time. However, for complex functions of higher dimension, this might change, that is, 60% of execution time is for evaluating the Shifted Schwefel's fitness function. Li *et al.* [8] have suggested a hardware software co-design method for implementing the PSO algorithm. We have observed that while optimising the fitness function like Shifted Schwefel's in the co-design platform, where fitness function is evaluated in the software and the remaining part of the algorithm is implemented in the hardware, the bus communication time is ∼106 ms. In contrast, if the complete DE algorithm along with fitness function is implemented in the hardware, it takes ∼128 ms. This concludes that bus communication overhead is dominating (i.e. 82.8%) the overall hardware execution time. This is observed when the embedded processor was operating at 200 MHz. To reduce the bus communication overhead, the total DE algorithm including the fitness function evaluation can be implemented in the embedded processor of the FPGA. This approach may result to a marginal acceleration in optimisation time. Both these approaches will not give any additional improvement in terms of the execution speed. So the alternate choice is to implement both the algorithm and fitness function evaluation in the hardware and use it as a dedicated coprocessor. For accelerating the computational intensive operations coprocessor based dedicated accelerators have been used [9].

In this work, a dedicated coprocessor for DE algorithm is developed and integrated with the embedded processor (PowerPC 440) to solve optimisation problems including spectrum allocation (SA) in cognitive radio network [10]. The proposed coprocessor is scalable in terms of the optimisation parameters, maximum number of iterations (G_{MAX}), population size (*NP*) and dimension (*D*). In the proposed design, both the fitness function evaluation and DE algorithm are in a single module rather than in two different modules. The software execution time of both arithmetic of DE algorithm is evaluated and compared with the hardware execution time of the algorithm.

The rest of the paper is organised into ten different sections. Section 2 presents existing literature about FPGA implementation of evolutionary algorithms. Section 3 presents brief introduction about DE algorithm and its software profiling is described in Section 4. The proposed hardware architecture for the DE algorithm is described in Section 5. Section 6 presents system on chip implementation of the DE coprocessor with auxiliary processor unit (APU) interface details. Section 7 presents the experimental setup. Section 8 describes results and analysis of DE coprocessor and Section 9 presents SA in cognitive radio as a case study of real-time application of DE coprocessor followed by conclusions in Section 10.

2 Related works

In literature, different evolutionary algorithms have been implemented using FPGA as shown in Table 1. A customised intellectual property (IP) of genetic algorithm was implemented in the Xilinx FPGA and integrated with PowerPC 405 processor based system on chip (SoC) and the speed enhancement up to 5.16× was achieved in Virtex-II Pro development kit [7]. A modular co-design architecture was developed for PSO algorithm [8], in which particle positions were updated in hardware whereas the fitness function was evaluated on a Nios-II embedded processor. Owing to this approach, the design had a flexibility to modify the fitness functions in the software depending on the applications. With this approach various embedded applications can be developed simply by changing the objective function. This design achieved speedup of 20× in Altera development kit. Hardware architecture of pipelined PSO (PPSO) was developed along with the parallel PSO (pPSO) framework which consists of multiple Nios-II processors using system-on-a-programmable-chip (SOPC) methodology and resulted speedup of 98× compared to the software implementation of the PSO algorithm in Altera development kit [11]. A modular, flexible and reusable multi-swarm PSO parallel hardware architecture was proposed to overcome the drawbacks of software implementation of the PSO algorithm using a freescale microcontroller and Xilinx MicroBlaze soft processor core [12]. A hardware accelerator for pPSO algorithm was reported and validated its performance by optimising test bench functions on MicroBlaze processor-based SoC in a Virtex-6 development kit [13]. Apart from the above works, different variants of

Table 1 Review of existing literature on FPGA implementation of evolutionary algorithms

Work	Algorithm	Processor freq, MHz	IP freq (max freq) in MHz	Speedup	Target Board
[7]	GA	PowerPC (200)	50 (50)	5.16×	Xilinx Virtex II Pro
[8]	PSO	Nios-II (50)	50 (50)	20×	Altera DE2–70
[11]	PSO	4 Nios-II (50)	50 (76.3)	98×	Altera Stratix
[12]	PSO	Freescale (25)	25 (–)	359× –653×	MC9S12DP256B
		MicroBlaze (25)	25 (42.5) and 25 (29.8)	37x–52x	Xilinx Virtex-II PRO &SP3E
[13]	PSO	MicroBlaze (200)	– (233)	18× –135×	Xilinx Virtex-6
[20]	floating point DE	PPC440 (200)	50 (120.6)	200 ×	Virtex-5FXT
proposed work	fixed point DE	PPC440 (200)	33 (65)	80× –150×	Virtex-5FXT

```
Algorithm 1
Step 1: Read the control parameter values of the DE algorithm : scale factor F, crossover rate CR, maximum number of iterations $G_{MAX}$ and the population size NP from user.
Step2: Set the generation number Gen=0 & randomly initialise population of NP individuals

  for i=1 to NP //do for each individual sequentially  do
    for j=1 to D //do for each individual sequentially  do
      $X_{i,j}^{(G)}$=$X_{min}$+($X_{max}$-$X_{min}$)*rand();//each individual uniformly distributed in the range $[X_{min}, X_{max}]$ where $X_{min}$=$\{x_1^{min}, x_2^{min}, ...., x_D^{min}\}$ and $X_{max}$=$\{x_1^{max}, x_2^{max}, ...., x_D^{max}\}$
    end for
  end for
  Step 3:
  while the stopping criterion is not satisfied OR Gen< $G_{MAX}$ do
    for i=1 to NP //do for each individual sequentially  do
      Step 3.1: Mutation Step
      Generate a mutant vector $V_i^{(G)}$=$\{v_{1,i}^G, ...., v_{D,i}^G\}$ corresponding to the ith target vector $X_i^{(G)}$
      via the differential mutation scheme of DE as:$V_i^{(G)} = X_{r_1}^{(G)} + F * (X_{r_2}^{(G)} - X_{r_3}^{(G)})$
      Vector indices $r_1$, $r_2$ and $r_3$ are randomly chosen, where $r_1$, $r_2$ and $r_3$ $\{1,...,NP\}$
      Step 3.2: Crossover Step
      Generate a trial vector $U_i^{(G)}$=$\{u_{1,i}^{(G)}, ...., u_{D,i}^{(G)}\}$ for the ith target vector $X_i^{(G)}$ through binomial crossover in the following way:
       if $(rand_{i,j}[0,1] \leq CR \; or \; j = j_{rand})$, then
      $u_{j,i}^{(G)} = v_{j,i}^{(G)}$
      else
      $u_{j,i}^{(G)}$=$x_{j,i}^{(G)}$
      endif
      Step 3.3: Selection Step
      Evaluate the trial vector $U_i^{(G)}$:  if $f(U_i^{(G)}) \leq f(X_i^{(G)})$, then
      $X_i^{(G+1)}$= $U_i^{(G)}$
      else
      $X_i^{(G+1)}$= $X_i^{(G)}$
      endif
    end for
    Step 3.4:Increase the Generation Count: Gen = Gen + 1
  end while
```

Fig. 1 *Pseudo-code for the differential evolution algorithm*

Table 2 Benchmark functions used for performance analysis

No.	Function name	Dimension	Optimal fitness value
Fun1	Rosenbrock	2	0
Fun2	Goldstein	2	3
Fun3	sphere	3	0
Fun4	variably dimensioned	4	0
Fun5	shifted sphere	32	0
Fun6	shifted Schwefel's	32	0

Table 3 Control parameters of the DE algorithm

Control parameters	Value
population size (*NP*)	8, 16, 32
total number of independent runs (G_{MAX})	1, 50, 100
dimension (*D*)	8, 16, 32
weighting factor (*F*)	0.9
crossover rate (CR)	0.9

PSO algorithms were implemented on a FPGA without addressing the acceleration of execution speed [14–19].

Recently the authors have proposed a floating point implementation of DE algorithm in the SoC, and reported that the DE IP core running at 50 MHz accelerates the execution speed approximately by 200× compared with its equivalent software implementation on a PowerPC 440 processor [20]. Since, there is no work reported in the literature which implements fixed point DE algorithm as a coprocessor suitable for embedded applications. In this work, we have developed a coprocessor for DE algorithm, interfaced with the PowerPC embedded processor and tested its performance by solving mathematical test bench functions and a practical SA problem.

3 DE algorithm

Basic DE algorithm has three major steps, (i) reading control parameter values, (ii) initialisation of population and (iii) mutation, crossover and selection process. The complete pseudo-code of DE algorithm is given in Algorithm 1 (see Fig. 1). The performance of DE coprocessor is tested by optimising a set of numerical test bench functions as tabulated in Table 2. These numerical functions (CEC 2005 [21] and CEC 2010 [22]) include four low dimension and two high dimension functions. The DE algorithmic control parameters are listed in Table 3.

4 Software profiling of DE algorithm

Software profiling of both the fixed and floating point DE algorithm is carried out on Xilinx PowerPC processor [23] with clock frequency set to 200 MHz. The profiling results of three computational intensive modules of the DE algorithm for different test functions with maximum generations $G_{\text{MAX}} = 1000$ and population size $\text{NP} = 8$ are tabulated in Table 4. From this table, it is observed that for Fun1 floating point DE algorithm takes 4721 ms time in contrast to fixed point DE, which takes 70 ms time for execution. This is because of the complexities of floating point arithmetic. In the embedded processor, floating point unit (FPU) is used for all the floating point arithmetic involved in the algorithm. From Table 4, it is also observed that all the test functions (except Fun6 function) takes less time for evaluation. The value inside the parenthesis refers to the % of total execution time that the particular module requires during execution.

Table 5 compares the average execution time (in millisec) and percentage of standard deviation (Std%) of execution time for both variants of DE algorithm implemented in the embedded processor (SW). The reported average and standard deviations are for 20 independent runs. This table shows results with maximum iterations $G_{\text{MAX}} = 1, 50, 100$ and for different population sizes $NP = 81, 16, 32$. This table reveals that for optimising high dimension test functions, fixed point software algorithm gives $\sim 4.96–7.36\times$ acceleration over the floating point software algorithm. For optimising low dimensional functions the execution time 32–48.45× is faster compared to the floating point algorithm.

5 Hardware architecture of DE algorithm

The architecture of the DE algorithm is shown in Fig. 2. It has seven modules, that is, memory initialisation, mutation, crossover, selection, random number generator, fitness evaluation and a control finite state machine (FSM) Module. The FSM is shown in Fig. 3. It has idle, initialisation, operation, waiting and reading states. In the idle state, all the modules are in the reset condition. In the initialisation state, FSM enables memory module when the inputs such as maximum number of generations (G_{MAX}), population size (*NP*) and dimension (*D*) are made available. During the operation state, control FSM enables internal modules according to the different stages of the algorithm, that is, crossover, mutation and selection. FSM will be in wait state until the execution of current module is completed then it will go to the next module for execution. In reading state, FSM reads the fitness value and writes into the output register.

Table 4 Profiling results of the software (SW) DE algorithm ($G_{\text{MAX}} = 1000$, $NP = 8$)

Test function	DE algorithm		Objective function		RNG		Float_operations
	SW float, ms	SW fixed, ms	SW float, ms	SW fixed, ms	SW float, ms	SW fixed, ms	SW float, ms
Fun1	50 (1%)	30 (43%)	10 (0.21%)	10 (14%)	40 (0.85%)	30 (43%)	4621 (97.88%)
Fun2	60 (1%)	30 (43%)	60 (0.81%)	10 (14%)	40 (0.54%)	30 (43%)	7228 (97.83%)
Fun3	90 (2%)	30 (38%)	10 (0.19%)	10 (13%)	50 (0.93%)	40 (50%)	5220 (97.20%)
Fun4	50 (1%)	40 (44%)	20 (0.26%)	20 (22%)	40 (0.51%)	30 (33%)	7674 (98.58%)
Fun5	1488 (5%)	1245 (69%)	294 (0.97%)	255 (14%)	520 (1.72%)	303 (17%)	27 944 (92.38%)
Fun6	1824 (6%)	1330 (29%)	3640 (11%)	2983 (64%)	570 (1.72%)	334 (7%)	27 042 (81.75%)

Table 5 Execution time of the DE algorithm implemented in software

Test function	G_{MAX}	$NP=8$			$NP=16$			$NP=32$		
		Float SW, ms (Std%)	Fixed SW, ms (Std %)	Acceleration factor	Float SW, ms (Std%)	Fixed SW, ms (Std%)	Acceleration factor	Float SW, ms (Std%)	Fixed SW, ms (Std%)	Acceleration factor
Fun1	1	4.91 (3.2)	0.15 (2.8)	32.73	9.44 (2.5)	0.26 (2.2)	36.31	18.36 (1.4)	0.52 (1.2)	35.31
	50	181.05 (1.4)	5.38 (0.9)	33.65	332.37 (0.7)	9.69 (0.4)	34.30	641.81 (0.4)	18.82 (0.2)	34.10
	100	363.17 (1.1)	10.38 (0.5)	34.99	673.21 (1.4)	19.33 (0.4)	34.83	1301.32 (1.1)	37.51 (0.2)	34.69
Fun2	1	8.01 (1.9)	0.18 (2.2)	44.50	15.02 (1.5)	0.31 (2.3)	48.45	28.73 (0.8)	0.61 (1.2)	47.10
	50	264.53 (1.4)	5.97 (0.9)	44.31	491.39 (0.9)	10.85 (0.4)	45.29	940.12 (0.7)	19.82 (0.2)	47.43
	100	536.05 (1.5)	11.96 (0.5)	44.82	994.24 (0.9)	21.64 (0.3)	45.94	1897.45 (0.7)	39.26 (0.2)	48.33
Fun3	1	5.12 (1.3)	0.16 (2.7)	32.00	10.13 (1.9)	0.31 (1.3)	32.68	19.88 (0.8)	0.61 (0.6)	32.59
	50	199.54 (0.8)	5.83 (0.6)	34.23	371.94 (0.4)	11.08 (0.3)	33.57	720.03 (0.2)	21.64 (0.2)	33.27
	100	397.91 (0.8)	11.62 (0.5)	34.24	740.14 (0.3)	22.08 (0.3)	33.52	1432.64 (0.2)	43.16 (0.2)	33.19
Fun4	1	9.99 (1.9)	0.23 (2.2)	43.43	19.36 (0.9)	0.45 (1.2)	43.02	38.38 (0.5)	0.84 (0.6)	45.69
	50	305.79 (0.6)	7.08 (0.4)	43.19	584.59 (0.3)	13.48 (0.2)	43.37	1145.25 (0.1)	26.46 (0.2)	43.28
	100	612.92 (0.7)	14.11 (0.3)	43.44	1178.46 (0.5)	26.94 (0.2)	43.74	2304.13 (0.3)	52.77 (0.2)	43.66
Fun5	1	41 (1.7)	6 (1.6)	6.83	81 (1.2)	11 (1.5)	7.36	162 (1.2)	23 (1.5)	7.04
	50	1132 (1.3)	207 (1.7)	5.47	2234 (2.1)	411 (1.6)	5.44	4439 (2.1)	809 (1.4)	5.49
	100	2254 (0.8)	412 (0.9)	5.47	4435 (0.9)	825 (2.1)	5.38	8809 (1.1)	1638 (2.1)	5.38
Fun6	1	85 (1.8)	15 (1.9)	5.67	170 (2.1)	30 (2.3)	5.67	339 (1.1)	62 (2.2)	5.47
	50	2251 (1.2)	446 (1.1)	5.05	4472 (1.5)	884 (1.7)	5.06	8916 (2.3)	1736 (2.1)	5.14
	100	4476 (1.3)	891 (2.1)	5.02	8745 (1.3)	1764 (1.1)	4.96	18 483 (0.9)	3537 (1.1)	5.23

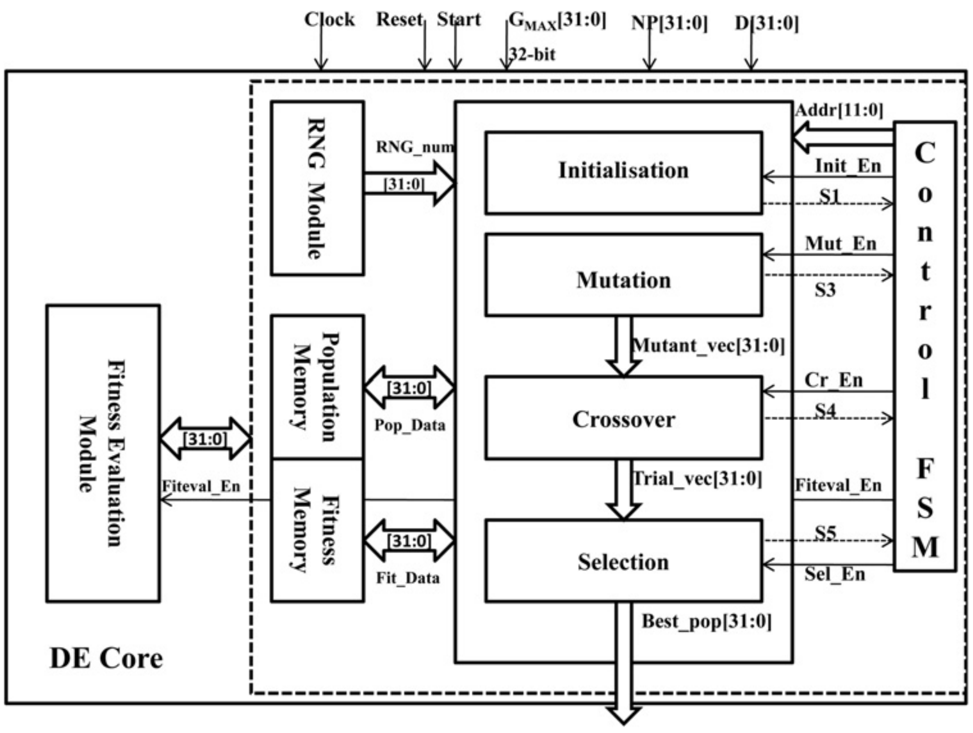

Fig. 2 *System architecture of DE algorithm*

5.1 *Initialisation module*

The architecture of initialisation module is shown in Fig. 4. The memory module has two separate memories, one is for storing the population values (population memory) and other is for storing their corresponding fitness function values (fitness memory). During the initialisation state, population values of all the particles (i.e. of size $NP \times D$) are randomly generated within the range of $[X_{\min}, X_{\max}]$, and stored in the population memory of size 4kbytes. The population values are accessed from the population memory by using a 12 bit address. Each population member is of

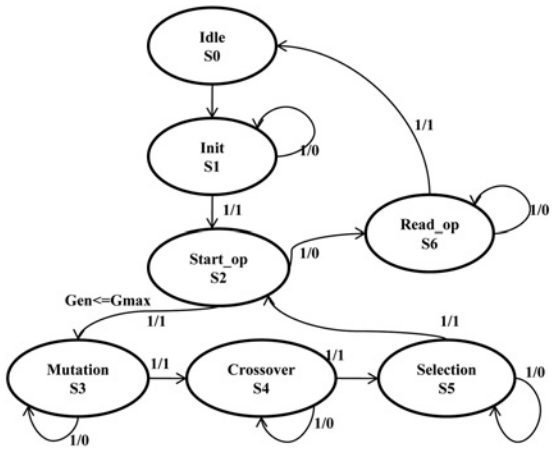

Fig. 3 *FSM diagram of DE algorithm*

dimension D (number of variables) and each variable is of size 32 bits. The maximum values of NP and D are set to 32. These values are input to the fitness evaluation module and after evaluating the fitness function the fitness values (of size 32 bit) are stored in the fitness memory of size 1 kbit. This process is repeated for all the population members.

5.2 Mutation module

After the population is initialised, mutation operation is performed by the mutation module. A mutant vector is generated for every target vector from the current population. In this module a mutant vector of size 128 bytes is generated for each population member. Three distinct vector indices r_1, r_2 and r_3 are generated in the range of 1 to NP by comparing the counter value with the value of multiplier. These indices are connected to the select lines of a multiplexer (MUX) unit. Three distinct target vectors each of size 1 kbits are obtained from the MUX unit as shown in Fig. 5. Then the mutation operation is performed by difference of any two of these selected three vectors scaled by a factor F and this difference is added to third one to obtain the mutant vector of size 256 bytes. A mutant vector is generated for all the population member of all dimensions.

5.3 Crossover module

The crossover operation is mainly responsible to increase the diversity among the mutant vectors. A trial vector is generated from the output of crossover module with a crossover probability CR as shown in Fig. 6. This crossover rate

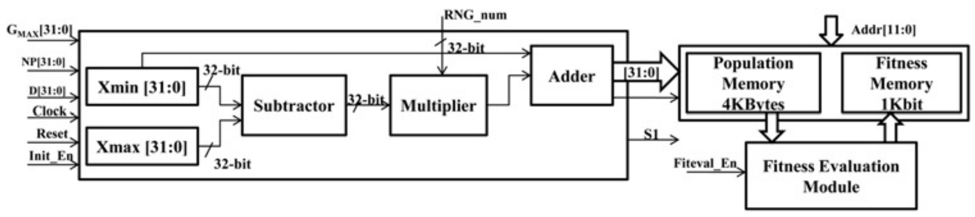

Fig. 4 *Initialisation module*

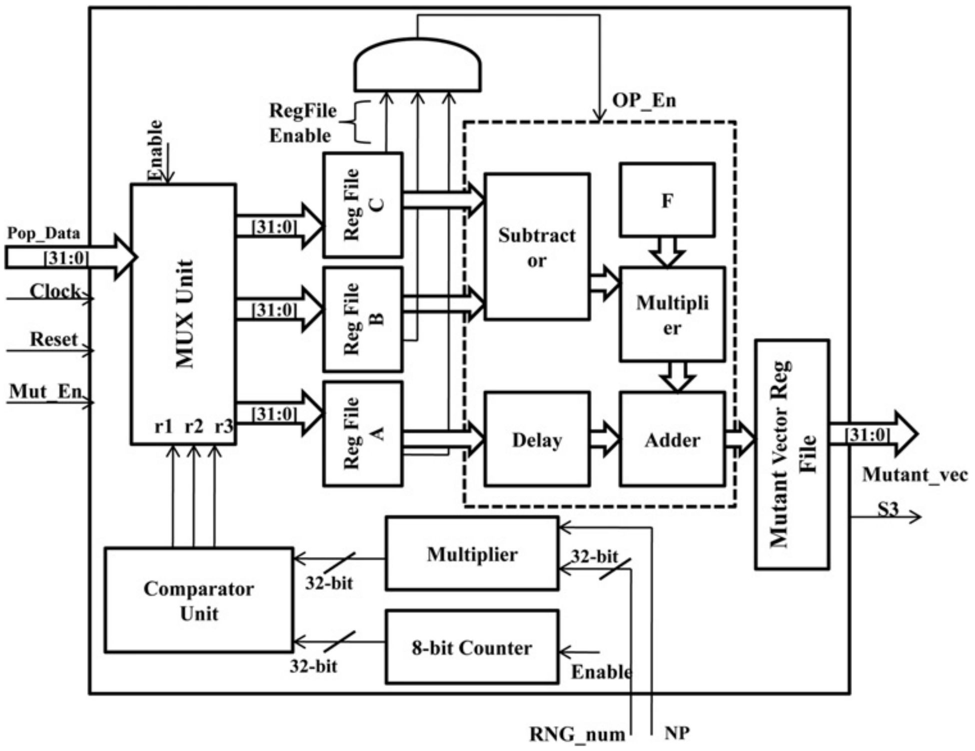

Fig. 5 *Mutation module*

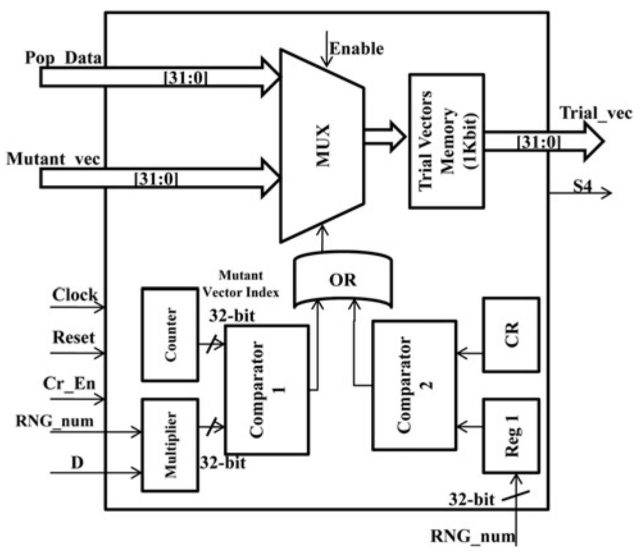

Fig. 6 *Crossover module*

controls the diversity of the population and helps the algorithm to escape from the local optima [1, 2], and ensures that the trial vector obtains at least one vector from the mutant vector. The register Reg1 has a random number stored in it. The output of Reg1 and CR are input to the comparator 2 module. The multiplier output and index of population member are input to the comparator Reg1. The output of both comparator 1 and 2 are input to a logic OR gate. The output of crossover module is either the mutant vector or the population vector as selected by the MUX unit.

5.4 Selection module

The output of crossover module is the trial vectors. These are input to the selection module as shown in Fig. 7. The fitness value of trial vector is evaluated by using the fitness evaluation module and if it is less than the fitness of the current population member then it selects the input as trial vector else the current population member is selected as the new population member. The output of MUX is the updated value of the current population memory. This process is repeated for all the iterations to improve the fitness of individuals and the process is stopped when the maximum number of generations is reached.

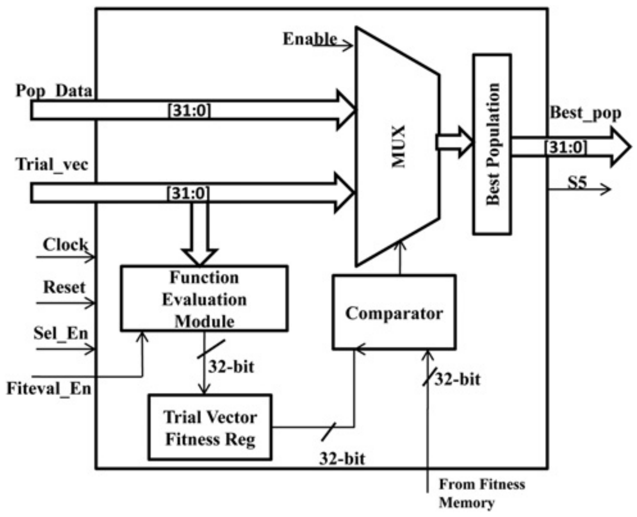

Fig. 7 *Selection module*

5.5 Fitness evaluation module

This module evaluates the fitness of each individuals. The fitness of each population and the population members of the complete population is evaluated and stored in the fitness memory. For different functions/applications, only the fitness module is modified.

5.6 RNG module

RNG module has great importance for the proper operation of the DE. Here, a linear feedback shift register (LFSR) is used for generating random numbers, as it is easy to implement and it produces fairly good pseudo-randomness. This module generates random numbers for the initial population module, selection module, crossover and mutation modules. The seed for random number generator is programmable and it is initialised to a non-zero value. If all zero value appears in the seed, then XOR operations continues to generate zeros and output becomes always zero. The architecture of 32 bit LFSR with maximum length polynomial $X^{32}+X^{22}+X^{2}+X^{1}+1$ is shown in Fig. 8. This module generates 2^{32}–1 random numbers.

6 Programmable system on chip (PSoC) implementation of the DE algorithm

PSoC is a programmable integrated system that has configurable processors, peripherals, memories, custom intellectual peripherals on a single FPGA. The proposed PSoC platform for implementing the DE algorithm is shown in Fig. 9. PowerPC 440 (PPC440) processor communicates with external peripherals such as double data rate synchronous dynamic random-access memory (DDRZ SDRAM), Block Ram memory (BRAM) controllers, universal asynchronous receiver/transmitter (UART) (RS-232), timer and interrupt controllers, joint test action group (JTAG) controller, clock generator via processor local bus (PLB). PPC440 is preferred over MicroBlaze processor because of its high speed of operation and efficient resource utilisation. DDR2 and BRAM controllers are used for storing heap and stack of program and data. UART is used for serial data transfer between the end user and processor. Timer and interrupt controllers are used for profiling the application. The clock generator provides necessary clock signals to all the modules and peripherals. USB JTAG controller is used to download the bitstream from host computer to FPGA board. PPC440 is directly coupled to the APU controller, which provides flexible high-bandwidth interface to DE coprocessor via fabric coprocessor bus (FCB). The coprocessor operates as an extension to the PowerPC. The APU interface details is shown in Fig. 10.

6.1 DE coprocessor with APU interface

APU interface allows the coprocessor to execute extended instruction set concurrently with PowerPC 440 embedded processor instructions set. It provides various coprocessor functions, such as a fully compliant PowerPC floating-point unit [23], or other custom function implementing algorithms appropriate for specific applications such as DE and PSO algorithms. The APU controller interface along

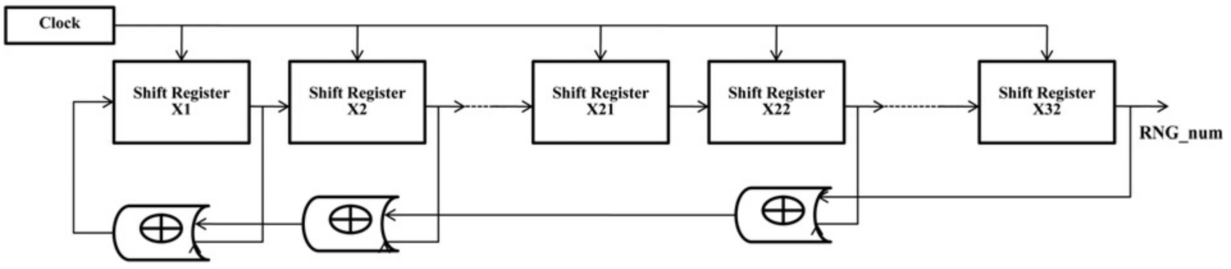

Fig. 8 *Random number generator*

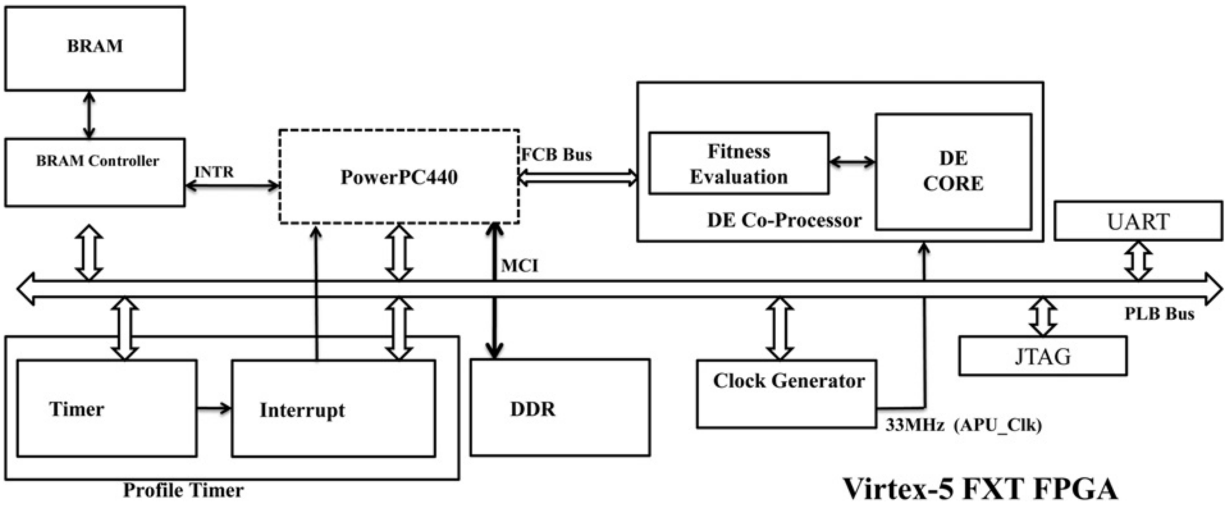

Fig. 9 *System on chip setup for DE algorithm*

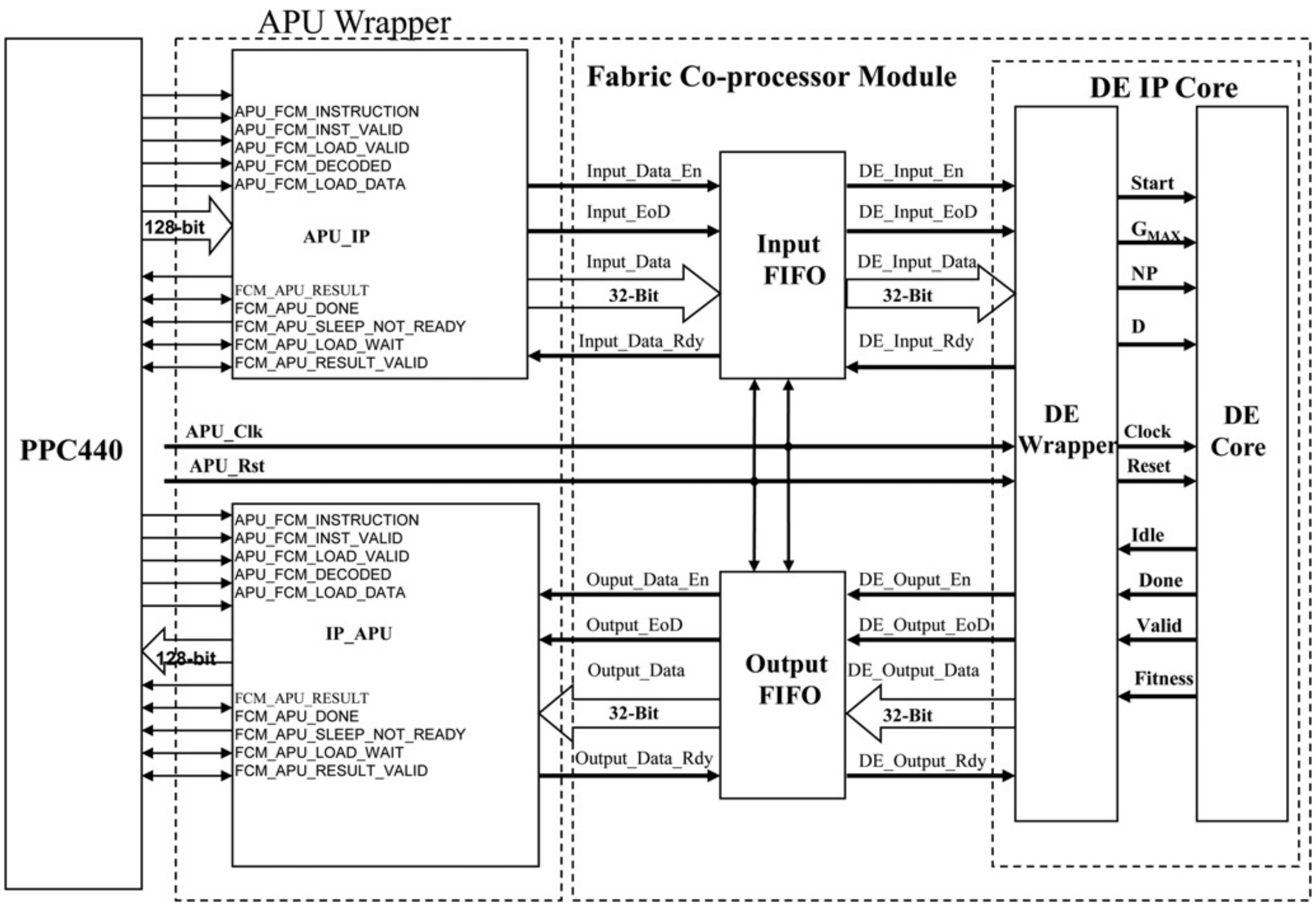

Fig. 10 *APU interface diagram of differential evolution IP core*

with fabric coprocessor module behaves as a coprocessor for PPC440. Since, the APU is independent of the processor to peripheral interface, it does not add an extra load to the PLB bus. The PPC440 supports three primary types of instructions to be used for APU [23]. In this work, load/store instructions are used for accessing the APU, in which maximum of 128 bits of data can be transferred in a single clock cycle or it can be transferred as four sets of 32 bits. The details of interfacing the DE IP core with the embedded processor is shown in Fig. 10. The FCB bus is specifically targeted to host the DE coprocessor without intervention of the processor instructions. The DE core frequency is adjusted by the clock generator and set to 33 MHz. However, it can be increased up to maximum frequency of IP core subject to maintaining the desired clock ratio of processor to APU controller.

Fig. 10 has two asynchronous first in, first out (FIFOs) (depth of four and width of 32 bits) interfaced at the input and output of the DE core. The input signal is processed as a stream and each stream has four samples and three of which are used for G_{MAX}, NP and D. The remaining sample is used for checking whether the FIFO is 50% full or not. In this architecture, ‘Output_Data’ and ‘Input_Data’ are two 32 bit width data buses for data input and output of the IP core, respectively. The working principle of the DE IP core is described as below.

1. PowerPC writes the input data G_{MAX}, NP and D in three clock cycles. The IP core receives data from the PowerPC, till the FIFO is full. This is ensured by the control signal ‘Input_EoD’. When the FIFO is 50% full ‘Input_EoD’ becomes logical high.
2. When the FIFO is 50% full, it will enable ‘DE_Input_En’ as logical high, and when the IP core is ready for processing it will give a handshaking signal ‘DE_Input_Rdy’ as logical high. The FIFO sends the data to the IP core till ‘DE_Input_EoD’ is logical high.
3. When the IP core processes only single sample on the stream, it gives ‘DE_Output_En’ as logical high and this is acknowledged by the output FIFO with handshaking signal ‘DE_Ouput_Rdy’. When this logical signal is high then the IP core sends the processed samples to the output FIFO till ‘DE_Output_EoD’ is high.
4. When the output FIFO is full, FIFO will send back the data to APU of PowerPC processor.

The APU wrapper contains two different modules namely IP_APU and APU_IP. The APU_IP module receives data from the processor and sends it to DE IP, whereas the IP_APU module receives the final solution from the DE IP core and sends it to the processor (PPC440). The APU_IP receives 128 bit signal, but the DE IP has only 32 bit width input, so the IP receives a full set of data in four clock cycles. Similarly the IP_APU module receives 128 bits of data from the IP in four clock cycles. The APU wrapper is interfaced with the IP core using six control signals ‘Input_Data_En, Input_Data_Rdy, Input_EoD, Output_Data_En, Output_Data_Rdy, Output_EoD’. A FSM with five states, that is, load, load_valid, store, store_valid and idle states control the data flow between Processor, IP_APU and APU_IP.

7 Experimental setup

In this work, the basic DE algorithm is considered for coprocessor implementation. The DE algorithmic parameters are tabulated in Table 3. The DE software code is ported into the PPC440 processor using 32 bit fixed and floating point C code, later algorithm is coded in Verilog language for implementing in the hardware. An IP core for DE algorithm is developed and simulated using Xilinx ISE 10.1, then a synthesisable IP core is developed and subsequently a coprocessor is designed for accelerating the DE algorithm. For functional verification, the wrapper logic and the DE core are simulated using a test bench with code coverage of 99.9% and the simulation results are shown in Fig. 11 for Fun6 with $G_{\mathrm{MAX}} = 1$, $NP = 8$ and $D = 4$. When DE_Output_Rdy signal is logic high, the resultant fitness value is available at DE_Output_Data port which is in fixed point format. After logic high on DE_Output_Rdy signal, DE_Input_Rdy is high because of scheduling for next set of G_{MAX}, NP and D values. From the results it is observed that the IP core consistently giving the same results.

The IP core frequency is set to 33 MHz and connected to PPC440 of Xilinx Virtex-5 FPGA using tightly coupled APU controller interface. The performance of the coprocessor is evaluated by optimising six numerical benchmark functions used in CEC 2005 and 2010 competitions [21, 22]. Owing to the empirical nature of DE algorithm, evolution parameters are subject to modification. In the proposed coprocessor, population size (NP), number of generations (G_{MAX}) and dimension (D) can be modified by the users through the embedded processor without redesigning the hardware.

8 Results and analysis

8.1 Timing results

Initially, the complete DE optimisation algorithm is ported into the PowerPC processor of Xilinx Virtex-5 FPGA for

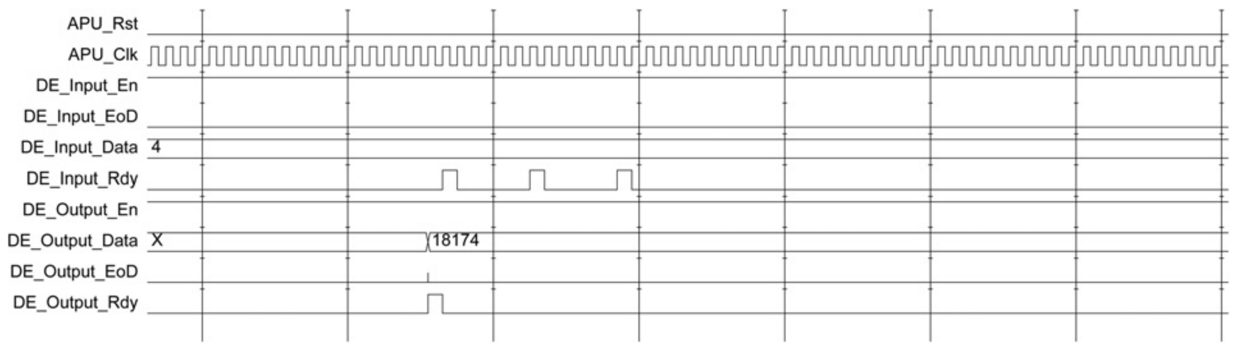

Fig. 11 *Functional simulation of DE IP core*

Table 6 Timing results of DE coprocessor and its acceleration factor (AF) over floating and fixed point software execution time

Test function	G_{MAX}	*NP* = 8			*NP* = 16			*NP* = 32		
		HW, ms (Std%)	AF float	AF fixed	HW, ms (Std%)	AF float	AF fixed	HW, ms (Std%)	AF float	AF fixed
Fun1	1	0.05 (1.1)	98.20	3.00	0.1 (1.4)	94.40	2.60	0.2 (1.0)	91.80	2.60
	50	2.13 (1.2)	85.00	2.53	3.9 0.7)	85.22	2.48	7.6 (0.2)	84.45	2.48
	100	4.27 (1.0)	85.05	2.43	8.21 (1.5)	82.09	3.94	15.2 (0.2)	85.61	2.47
Fun2	1	0.05 (1.5)	160.20	3.60	0.1 (1.3)	150.20	3.10	0.2 (1.4)	143.65	3.05
	50	2.23 (1.2)	118.62	2.68	4.06 (0.5)	121.03	2.67	7.8 (0.3)	120.53	2.54
	100	4.43 (0.7)	121.00	2.70	8.11 (0.5)	122.59	2.67	15.6 (0.2)	121.63	2.52
Fun3	1	0.07 (1.1)	73.14	2.29	0.13 (1.8)	77.92	2.38	0.26 (1.3)	76.46	2.35
	50	2.6 (1.1)	76.75	2.24	4.9 (0.5)	75.91	2.26	9.57 (0.3)	75.24	2.26
	100	5.3 1.1)	75.08	2.19	9.9 (0.6)	74.76	2.23	19.06 (0.2)	75.16	2.26
Fun4	1	0.08 (1.6)	124.88	2.88	0.15 (1.9)	129.07	3.00	0.3 (0.9)	127.93	2.80
	50	2.9 (1.2)	105.44	2.44	5.4 (0.7)	108.26	2.50	10.5 (0.3)	109.07	2.52
	100	5.8 (1.2)	105.68	2.43	10.9 (0.7)	108.12	2.47	21.1 (0.6)	109.20	2.50
Fun5	1	0.5 (0.8)	82.00	12.00	0.8 (0.3)	101.25	13.75	1.6 (0.2)	101.25	14.38
	50	11.9 (0.2)	95.13	17.39	23.5 (0.1)	95.06	17.49	46.6 (0.1)	95.26	17.36
	100	23.7 (0.1)	95.11	17.38	46.7 (0.1)	94.97	17.67	92.6 (0.1)	95.13	17.69
Fun6	1	0.6 (0.6)	141.67	25.00	1.2 (0.3)	141.67	25.00	2.3 (0.2)	147.39	26.96
	50	16.4 (0.4)	137.26	27.20	32.4 (0.1)	138.02	27.28	64.5 (0.1)	138.23	26.91
	100	32.6 (0.5)	137.30	27.33	64.4 (0.1)	135.79	27.39	128 (0.1)	144.40	27.63

software implementation, then the complete DE algorithm is executed using the DE coprocessor. The execution time of the DE algorithm for different population sizes (8, 16, 32) and for three different generations (1, 50, 100) is evaluated for 20 independent runs. The average execution time of the algorithm using the coprocessor is tabulated in Table 6 and this is referred as hardware (HW) time. The acceleration factor (AF) of the coprocessor with respect to software floating and fixed point execution time are tabulated as AF (float) and AF (fixed), respectively. The values in parenthesis refer to the percentage of standard deviation of execution time. From this table, it is observed that the coprocessor execution time is up to 73.14–160.20× faster than the software execution time for floating point DE algorithm. In contrast, it is only 2.19–27.63× faster compared with fixed point DE algorithm. Further it is observed that for lower dimension functions coprocessor acceleration AF (fixed) is small as compared to higher dimension functions. This table also reveals that the execution time of HW coprocessor for different functions is scaling up with the population size and maximum number of generations. A comparison of the average speedup of floating to fixed, floating to hardware and fixed to hardware implementations for different benchmark function (G_{MAX} = 100 and *NP* = 8) are illustrated in Fig. 12.

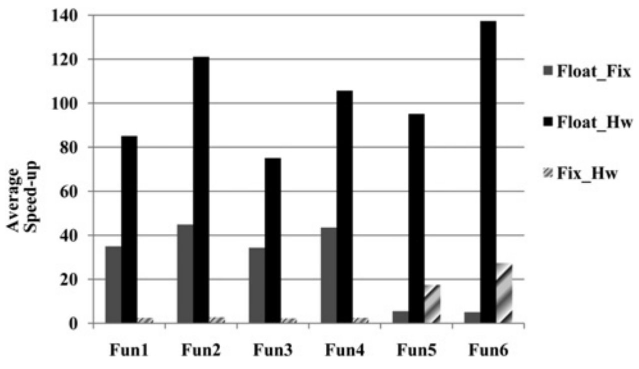

Fig. 12 *Average speedup for benchmark functions*

8.2 Synthesis results

The hardware IP is designed with multiple modules using Verilog language and the code size is ~1000 lines. It is parameterised in terms of DE population size (*NP*), dimension (*D*) and maximum number of generation G_{MAX}. Table 7 shows XST (Xilinx Synthesis Tool) synthesis results (resource utilisation) for optimising different benchmark functions with population size *NP* = 32. The targeted FPGA is Xilinx Virtex-5 XC5VFX70T. It has several device primitives like BRAM, DSP48E, Slices and LUTs. Each BRAM is of 36 kbits size. It can be configured as two separate memories of 18 kbits size each. DSP48E slice is a digital signal processing logic element and it can perform multiply-accumulator, multiply-adder, one or *n*-step counter along with logic operations such as AND, OR and XOR. Slices are combination of LUTs and flip flops, used for implementing the digital logic of desired IP. For higher dimensional test bench functions 6% Block RAM (BRAM) is utilised compared with other functions. The resource utilisation for the Fun2 (60% of DSP48E, 7% of Slice

Table 7 Resource utilisation

Test function	BRAM	DSP48E	Slice registers	Slice LUTs	Slices	LUT FF pairs	Max freq, MHz
Fun1	3 (2%)	61 (47%)	2888 (6%)	3936 (8%)	1586 (14%)	1711 (33%)	39.33
Fun2	3 (2%)	77 (60%)	3150 (7%)	7315 (16%)	2546 (22%)	2152 (25%)	36.04
Fun3	3 (2%)	20 (15%)	3097 (6%)	3928 (8%)	1522 (14%)	2031 (40%)	67.04
Fun4	3 (2%)	61 (47%)	2883 (6%)	4065 (9%)	1625 (14%)	1688 (32%)	60.56
Fun5	10 (6%)	42 (32%)	2849 (6%)	3667 (8%)	1317 (11%)	1890 (40%)	64.67
Fun6	10 (6%)	41 (32%)	2886 (6%)	3753 (8%)	1485 (13%)	1682 (33%)	64.72
SA	84 (56%)	70 (54%)	4892 (10%)	7371 (16%)	2968 (26%)	2731 (28%)	63.55

registers, 16% of LUTs and 22% of Slices) is high compared with other functions due to its computational complexity.

9 Real-time application as a case study: SA in cognitive radio

In the current wireless communication domain, spectrum scarcity is because of the rigid licensing policy [24]. Dynamic SA is an alternative to overcome this problem. Cognitive radio is the future technology which supports the dynamic SA [25]. In the cognitive radio domain there are two different type of users (a) primary user or licensed user and (b) secondary user or unlicensed user. A primary user has the priority to use an allotted spectrum band, however, in the absence of primary user, a secondary user can access the same band till a primary user demands for it. In the distributed network architecture, each secondary user determines the spectrum availability and allocate the desired spectrum. In this scheme, a secondary user considers the locally available information from the neighbourhood users and decides spectrum assignment.

As each secondary user implicitly have an embedded computing platform, the SA task can be performed by it. However, running the SA on an embedded processor consumes most of the platform resources, thereby degrading the performance of other applications running on it. Hence, there is a requirement for a dedicated hardware peripheral for performing the SA task. This is the motivation for choosing this application as a case study in this work. This problem is posed in [10] and have been solved by using genetic, quantum genetic and PSO algorithms. In this paper, the same problem is solved by using the developed DE hardware coprocessor regarding execution speedup and acceleration factor.

The general SA model consists of a channel availability matrix ($\boldsymbol{L}$) representing the channel availability, $\boldsymbol{L} = \{l_{n,m} | l_{n,m} \in \{0,\ 1\}\}_{N\times M}$, where $l_{n,m} = 1$ if and only if channel m is available to user n, else $l_{n,m} = 0$, channel reward matrix ($\boldsymbol{B}$) representing the channel reward, $\boldsymbol{B} = \{b_{n,m}\}\ N \times M$, where $b_{n,m}$ represents the reward that can be obtained by the user n that uses channel m, and an interference constraint matrix ($\boldsymbol{C}$) representing the interference constraints among the secondary users (n and p), $\boldsymbol{C} = \{c_{n,p,m} | c_{n,p,m}$ belongs to $\{0,\ 1\}\}_{N\times N\times M}$, where $c_{n,p,m} = 1$ if both the secondary users n and p use the channel m simultaneously else $c_{n,p,m} = 0$ [10]. The required solution is a conflict free channel assignment matrix $\boldsymbol{A} = \{a_{n,m} | a_{n,m}$ belongs to $\{0,\ 1\}\}_{N\times M}$, where $a_{n,m} = 1$ if channel m is allocated to secondary user n, else $a_{n,m} = 0$ [10, 26]. In this work, the reward matrix and constraint matrix are initialised as [26].

In real time applications, users perform network-wide SA operation faster than the change in spectrum environment. In this work, the assumption is that the location, available spectrum etc. are static, thus $\boldsymbol{L}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ remains constant during a particular allocation period. As the SA model can be inherently seen as an optimisation problem, so the DE algorithm is proposed to solve the allocation problem. The proposed architecture for DE algorithm is exploited to select the appropriate channel for secondary users from the available channels without interfering with the primary users. The conflict free spectrum assignment matrix $\boldsymbol{A}$ must satisfy the interference constraints defined by $\boldsymbol{C}$

$$a_{n,m}a_{p,m} = 0, \quad \text{if} \quad c_{n,p,m} = 1,\ \forall 1 \le n,\ p \le N,\ 1 \le m \le M \tag{1}$$

The above equation states that if the constraint $c_{n,p,m} = 1$ then one of the secondary user between n and p can use the channel m depending on the reward value of the user. If the user n has more reward than user p, then the channel m will be used by the user n and vice versa. For the given $\boldsymbol{L}$ and $\boldsymbol{C}$, the objective of SA is to obtain the conflict free channel assignment matrix by maximising the reward sum $U(R)$. Thus the optimal conflict free channel assignment matrix $\boldsymbol{A}^*$ is selected from the set of conflict free channel assignment for a given set of N users and M spectrum bands and constraints $\boldsymbol{C}$ as shown in (2)

$$\boldsymbol{A}^* = \underset{A \in \wedge(\boldsymbol{L},\boldsymbol{C})_{N,M}}{\arg\max}\ U(R) \tag{2}$$

For improving the efficiency of SA one or more fitness functions need to optimised. In this work, maximum sum reward (MSR) is considered as the fitness function to validate the hardware framework. MSR is defined as [10]

$$\text{MSR}: U(R) = \sum_{n=1}^{N}\sum_{m=1}^{M} a_{n,m}b_{n,m} \tag{3}$$

In the proposed SA scheme, each population specifies a possible conflict free channel assignment matrix. To decrease the search space, we propose to encode only the elements that corresponds to $l_{n,m} = 1$. The length of the population is equal to the number of elements equal to 1 in the $\boldsymbol{L}$. The value of every element in the population is randomly generated that satisfies interference constraints $\boldsymbol{C}$.

The proposed DE-based SA algorithm proceeds as follows:

1. Given $\boldsymbol{L} = \{l_{n,m} | l_{n,m} \in \{0,\ 1\}\}_{N\times M}$, $\boldsymbol{B} = \{b_{n,m} | b_{n,m} \in \{0,\ 1\}\}_{N\times M}$ and $\boldsymbol{C} = \{c_{n,p,m} | c_{n,p,m} \in \{0,\ 1\}\}_{N\times N\times M}$, set the dimension of the population as $D = \sum_{n=1}^{N}\sum_{m=1}^{M} l_{n,m}$, and set $\boldsymbol{L}_1 = \{(n,\ m) | l_{n,\ m} = 1\}$ such that the elements in L_1 are arranged in ascending order with n and m.
2. Randomly generate the initial population $\boldsymbol{X}_i = [x_{1,i},\ ....,\ x_{3,i}, ..., x_{D,i}]$ where $x_{d,i} \in 0, 1, i \in (1 \ldots NP)$ and $d \in (1, \ldots, D)$.
3. Map the population $x_{d,i}$ to $a_{n,m}$, where $(n,\ m)$ is the dth element of $\boldsymbol{L}_1$ for all $d \in 1,\ \ldots,\ D$ and $i \in (1,\ \ldots,\ NP)$. The complete $\boldsymbol{A}$ matrix should satisfy the constraint matrix $\boldsymbol{C}$, if any violations are there then one of the user will get the channel m depending on their reward value and the corresponding element of the matrix $\boldsymbol{A}$ is set to 1 or 0.
4. Compute the fitness of the each individual of the current population.
5. Carry out the mutation, crossover, selection and update the population as defined in Algorithm 1 (see Fig. 1).
6. If it reaches the predefined maximum generation then derive the assignment matrix as mentioned in the step 3 and stop the process else go to step 3 and continue.

Both the fitness function and DE algorithm are evaluated in the coprocessor. Here, the algorithmic parameters (G_{MAX}, NP) and the SA parameters such as number of secondary users N, number of channels M and number of primary users K are parameterised and can be changed through the embedded processor. The execution time for evaluating the SA both in the embedded processor (software) and in the coprocessor is executed for 20 independent runs. Table 8 shows the software execution and acceleration factor for both arithmetic implementation. Table 9 shows the coprocessor execution time (HW) and acceleration factor w.r.to both arithmetic of algorithms executed in the processor.

Table 8 Execution time of SA problem in software

Test function	G_{MAX}	*NP* = 8			*NP* = 16			*NP* = 32		
		Float SW, ms (Std%)	Fixed SW, ms (Std%)	Acceleration factor	Float SW, ms (Std%)	Fixed SW, ms (Std%)	Acceleration factor	Float SW, ms (Std%)	Fixed SW, ms (Std%)	Acceleration factor
MSR(5 × 5 × 5)	1	42 (0.5)	2.2 (0.8)	19.09	82 (0.7)	4.3 (0.8)	19.07	163 (0.4)	8.3 (0.5)	19.64
	50	1068 (0.3)	63 (0.3)	16.95	2011 (0.3)	125 (0.4)	16.09	4113 (0.1)	248 (0.3)	16.58
	100	2008 (0.2)	127 (0.2)	15.81	4052 (0.1)	251 (0.3)	16.14	7820 (0.1)	497 (0.2)	15.73
	300	5786 (0.4)	381 (0.1)	15.19	11 276 (0.4)	757 (0.1)	14.90	22 364 (0.2)	1498 (0.1)	14.93
MSR(10 × 10 × 10)	1	150 (0.3)	10.1 (0.8)	14.85	296 (0.5)	19 (0.7)	15.58	589 (0.2)	37 (0.4)	15.92
	50	4286 (0.3)	266 (0.6)	16.11	8583 (0.3)	524 (0.3)	16.38	17 083 (0.2)	1034 (0.2)	16.52
	100	8326 (0.1)	536 (0.4)	15.53	16 598 (0.3)	1059 (0.4)	15.67	35 012 (0.5)	2074 (0.2)	16.88
	300	24 868 (0.2)	1614 (0.1)	15.41	49 823 (0.2)	3216 (0.2)	15.49	100 154 (0.3)	6335 (0.2)	15.81
MSR(20 × 20 × 20)	1	654 (0.5)	53 (0.4)	12.34	1287 (0.3)	100 (0.3)	12.87	2586 (0.7)	192 (0.4)	13.47
	50	15 028 (0.4)	1298 (0.4)	11.58	29 925 (0.2)	2586 (0.1)	11.57	58 974 (0.2)	5140 (0.1)	11.47
	100	29 890 (0.2)	2614 (0.4)	11.43	60 120 (0.3)	5135 (0.3)	11.71	120 036 (0.4)	10 161 (0.1)	11.81
	300	91 086 (0.2)	7906 (0.3)	11.52	180 210 (0.1)	15 583 (0.4)	11.56	359 860 (0.3)	31 189 (0.2)	11.54

Table 9 Execution time of SA problem in coprocessor

Test function	G_{MAX}	*NP* = 8			*NP* = 16			*NP* = 32		
		HW, ms (Std%)	AF float	AF fixed	HW, ms (Std%)	AF float	AF fixed	HW, ms (Std%)	AF float	AF fixed
MSR(5 × 5 × 5)	1	0.4 (0.3)	105.00	5.50	0.8 (0.7)	102.50	5.38	1.6 (0.4)	101.88	5.19
	50	11 (0.3)	97.09	5.73	22 (0.3)	91.41	5.68	43 (0.2)	95.65	5.77
	100	22 (0.2)	91.27	5.77	43 (0.2)	94.23	5.84	85 (0.2)	92.00	5.85
	300	66 (0.1)	87.67	5.77	129 (0.1)	87.41	5.87	254 (0.1)	88.05	5.90
MSR(10 × 10 × 10)	1	1.7 (0.5)	88.24	5.94	3.4 (0.3)	87.06	5.59	6.7 (0.2)	87.91	5.52
	50	44 (0.4)	97.41	6.05	87 (0.2)	98.66	6.02	172 (0.1)	99.32	6.01
	100	87 (0.3)	95.70	6.16	172 (0.3)	96.50	6.16	340 (0.1)	102.98	6.10
	300	262 (0.1)	94.92	6.16	515 (0.2)	96.74	6.24	1012 (0.1)	98.97	6.26
MSR(20 × 20 × 20)	1	8.1 (0.3)	80.74	6.54	15.6 (0.2)	82.50	6.41	30.5 (0.2)	84.79	6.30
	50	194 (0.2)	77.46	6.69	385 (0.1)	77.73	6.72	768 (0.1)	76.79	6.69
	100	385 (0.4)	77.64	6.79	762 (0.1)	78.90	6.74	1518 (0.1)	79.08	6.69
	300	1156 (0.2)	78.79	6.84	2277 (0.2)	79.14	6.84	4512 (0.1)	79.76	6.91

Table 10 Execution time (in Mega Clock cycles) for SA problem in embedded processor (SW) and Coprocessor (HW)

Test function	G_{MAX}	*NP* = 8			*NP* = 16			*NP* = 32		
		Float SW	Fixed SW	HW	Float SW	Fixed SW	HW	Float SW	Fixed SW	HW
MSR(5 × 5 × 5)	1	8.41	0.44	0.08	16.4	0.86	0.16	32.62	1.66	0.32
	50	213.6	12.6	2.2	402.2	25.0	4.4	822.6	49.6	8.6
	100	401.6	25.4	4.4	810.4	50.2	8.6	1564.1	99.4	17.0
	300	1157.2	76.2	13.2	2255.2	151.4	25.8	4472.8	299.6	50.8

The value parenthesis refers to the % of standard deviation. In these tables MSR ($N \times M \times K$) corresponds to the maximum sum reward for N number of secondary users, M number of channels and K number of primary users.

From Table 8, it is observed that fixed point software implementation gains acceleration of 11.43–19.64× over floating point implementation. Table 9 shows that the proposed DE coprocessor processing speed is ~5.19–6.91 × faster than fixed point software implementation and 76.79–105 × faster then floating point software implementation in the embedded processor (PPC 440). Table 10 tabulates execution time in terms of Mega clock cycles for optimizing MSR (5 × 5 × 5) objective function. Fig. 13 shows the comparison of the average speedup of

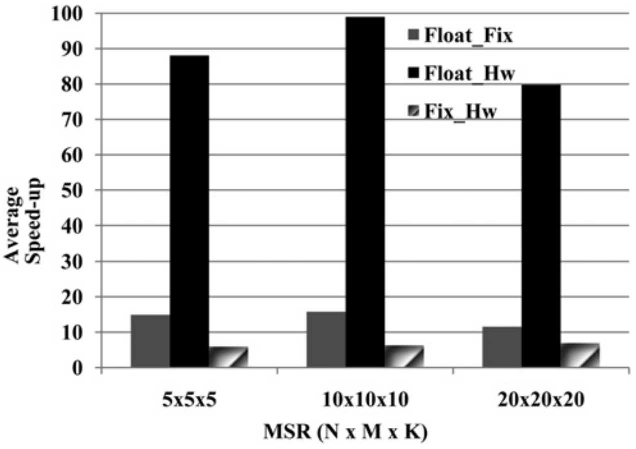

Fig. 13 *Average speedup for SA problem*

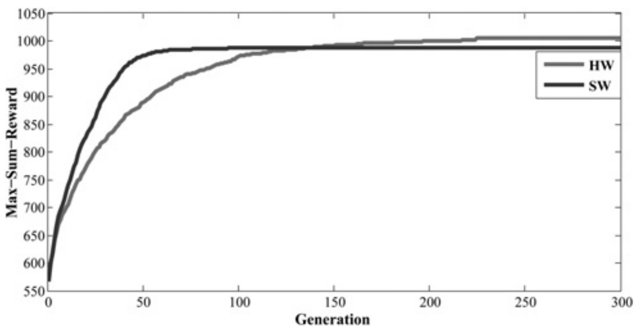

Fig. 14 *Comparison of the convergence graph*

floating to fixed, floating to hardware and fixed to hardware implementations of SA problem with $G_{\mathrm{MAX}} = 300$ and $NP = 32$ for three MSR ($N \times M \times K$) objective functions. It is observed that AF for floating to fixed is ∼11.53–15.8× because of high computational complexity in both arithmetic, but AF for coprocessor is 79.75–98.96× for floating and 5.89–6.91× over fixed arithmetic because of faster execution speed.

The algorithm is run for 20 independent runs with $N = 10$, $M = 10$ and $K = 10$ and the convergence graph is shown in Fig. 14. In this graph, SW means result obtained using PowerPC processor and HW means result obtained using the DE coprocessor. Initially there is some difference between the SW and HW results because of random number generation in the hardware, but after some iterations both attains almost same value. The curve shows that as higher the reward value, the user will be alloted a fair spectrum band. Table 11 tabulates the minimum, maximum, average, standard deviation and percentage of standard deviation of fitness value.

10 Conclusions

In this paper, we have proposed a scalable coprocessor with APU interface for accelerating the execution speed of the DE algorithm and it was implemented in a Xilinx Virtex-5 FPGA. To avoid the bus overhead, the complete DE algorithm with fitness function was implemented in the hardware instead of partitioning the design into software and hardware. To validate the performance of the coprocessor, firstly, six numbers of test-bench functions were optimised, then a practical problem of SA was solved using the coprocessor. For validation of the proposed framework the execution time for fixed point and floating point software implementation of DE algorithm is compared while optimizing test bench function and SA problem. The experimental results revealed that the software implementation of fixed point DE algorithm accelerated the execution speed by approximately 43.19–45.69× while optimising less complex test function (Fun4) and by 4.96–5.67× while optimising the 32 dimension test function (Fun6), as compared to the floating point D algorithm implemented in the embedded processor. The fixed point DE algorithm, along with the fitness evaluation, was also implemented in the coprocessor and the experimental results shown that an acceleration of approximately by 25–27.63× and 135.79–147.39× is attained while optimising a 32 dimension Fun6 complex test function compared to the fixed and floating point software implementation respectively. For optimising less complex fitness functions like Fun1, the coprocessor attained speedup of approximately by 2.43–3.94× over fixed point and 82.09–98.20× over floating point software implementation respectively. At the same time it was also observed that for SA problem, the coprocessor attained an acceleration of ∼76.79–105× and 5.19–6.91× compared to the floating point and fixed point point implementation of the algorithm in embedded processor, respectively. The proposed framework can be extended for accelerating other evolutionary techniques and can be used for designing Evolvable Hardware.

11 Acknowledgment

The authors thank the University Grants Commission, Government of India for providing necessary support.

12 References

1 Storn, R., Price, K.: ‘Differential evolution a simple and efficient heuristic for global optimization over continuous spaces’, *J. Global Optim.*, 1997, **11**, (4), pp. 341–359
2 Das, S., Suganthan, P.: ‘Differential evolution: a survey of the state-of-the-art’, *IEEE Trans. Evol. Comput.*, 2011, **15**, (1), pp. 4–31
3 Hay, J., Loo, K.: ‘Fast motion estimation using evolutionary strategy search algorithm’. Int. Conf. Digital Telecommunications, (ICDT’06), August 2006, pp. 16

Table 11 Statistical results of quality of solution using the processor (SW) and coprocessor (HW)

Test function	Min		Max		Avg		(Std%)	
	SW	HW	SW	HW	SW	HW	SW	HW
MSR(5 × 5 × 5)	280	290	280	290	280	290	0	0
MSR(10 × 10 × 10)	914	942	1022	1047	986	1008	2.34	2.11
MSR(20 × 20 × 20)	2547	2614	2985	3034	2753	2847	3.14	2.87

4 Pan, S.T.: 'Evolutionary computation on programmable robust iir filter pole-placement design', *IEEE Trans. Instrum. Meas.*, 2011, **60**, (4), pp. 1469–1479

5 Sekanina, L.: 'From implementations to a general concept of evolvable machines'. Proc. Sixth European Conf. Genetic Programming, (ser. EuroGP'03), 2003, pp. 424–433

6 Abdelfatah, W.F., Georgy, J., Iqbal, U., Noureldin, A.: 'FPGA-based real-time embedded system for RISS/GPS integrated navigation', *Sensors*, 2011, **12**, (1), pp. 115–147

7 Fernando, P.R., Katkoori, S., Keymeulen, D., Zebulum, R., Stoica, A.: 'Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine', *IEEE Trans. Evol. Comput.*, 2010, **14**, (1), pp. 133–149

8 Li, S.-A., Hsu, C.-C., Wong, C.-C., Yu, C.-J.: 'Hardware/software co-design for particle swarm optimization algorithm', *Inf. Sci.*, 2011, **181**, (20), pp. 4582–4596

9 Zicari, P., Corsonello, P., Perri, S., Cocorullo, G.: 'A matrix product accelerator for field programmable systems on chip', *Microprocess. Microsyst.*, 2008, **32**, (2), pp. 53–67

10 Zhao, Z., Peng, Z., Zheng, S., Shang, J.: 'Cognitive radio spectrum allocation using evolutionary algorithms', *IEEE Trans. Wirel. Commun.*, 2009, **8**, (9), pp. 4421–4425

11 Farmahini-Farahani, A., Vakili, S., Fakhraie, S.M., Safari, S., Lucas, C.: 'Parallel scalable hardware implementation of asynchronous discrete particle swarm optimization', *Eng. Appl. Artif. Intell.*, 2010, **23**, (2), pp. 177–187

12 Tewolde, G.S., Hanna, D.M., Haskell, R.E.: 'A modular and efficient hardware architecture for particle swarm optimization algorithm', *Microprocess. Microsyst.*, 2012, **36**, (4), pp. 289–302

13 Rogrio, M.C., Nedjah, N., Mourelle, L.M.: 'A hardware accelerator for particle swarm optimization' Applied Soft Computing, 2013 DOI 10.1016/j.asol.2012.12.034

14 Lin, C.-J., Tsai, H.-M.: 'FPGA implementation of a wavelet neural network with particle swarm optimization learning', *Math. Comput. Model.*, 2008, **47**, (910), pp. 982–996

15 Cavuslu, M.A., Karakuzu, C., Karakaya, F.: 'Neural identification of dynamic systems on FPGA with improved PSO learning', *Appl. Soft Comput.*, 2012, **12**, (9), pp. 2707–2718

16 Vasumathi, B., Moorthi, S.: 'Implementation of hybrid ANN – PSO algorithm on FPGA for harmonic estimation', *Eng. Appl. Artif. Intell.*, 2012, **25**, (3), pp. 476–483

17 Munoz, D., Llanos, C., Coelho, L., Ayala-Rincon, M.: 'Hardware particle swarm optimization based on the attractive-repulsive scheme for embedded applications'. Proc. Int. Conf. Reconfigurable Computing and FPGAs, December 2010, pp. 55–60

18 Munoz, D., Llanos, C., Coelho, L., Ayala-Rincon, M.: 'Hardware architecture for particle swarm optimization using floating-point arithmetic'. Proc. Ninth Int. Conf. Intelligent Systems Design and Applications, December 2009, pp. 243–248

19 Munoz, D., Llanos, C., Coelho, L., Ayala-Rincon, M.: 'Hardware particle swarm optimization with passive congregation for embedded applications'. Proc. VII Southern Conf. Programmable Logic (SPL), April 2011, pp. 173–178

20 Anumandia Kiran, K., Peesapati, R., Sabat, S.L., Udgata, S.K.: 'SoC based floating point implementation of differential evolution algorithm using FPGA', *Des. Autom. Embedded Syst.*, 2013, pp. 1–20, DOI 10.1007/s10617-013-9107-4

21 Suganthan, P.N., Hansen, N., Liang, J.J., *et al.*: 'Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization'. Technical report, Nanyang Technological University, Singapore, 2005

22 Tang, K., Li, X., Suganthan, P.N., Yang, Z., Weise, T.: 'Benchmark functions for the CEC'2010 special session and competition on large-scale global optimization'. Technical report, University of Science and Technology of China (USTC), School of Computer Science and Technology, Nature Inspired Computation and Applications Laboratory (NICAL): China, 2010

23 Xilinx: 'Reference Guide UG200 – embedded processor block in Virtex-5 FPGAs'. Technical report 10.1.3Xilinx, San Jose, California, 95124-3400, 2008

24 Mchenry, M.: 'Spectrum white space measurements'. Technical report, New America Foundation Broadband Forum, June 2003

25 Haykin, S.: 'Cognitive radio: brain-empowered wireless communications', *IEEE J. Sel. Areas Commun.*, 2005, **23**, (2), pp. 201–220

26 Peng, C., Zheng, H., Zhao, B.Y.: 'Utilization and fairness in spectrum assignment for opportunistic spectrum access', *Mob. Netw. Appl.*, 2006, **11**, (4), pp. 555–576