A Population-oriented Architecture for Particle Swarms

Jorge Peña
Université de Lausanne - UNIL
Institut de Mathématiques Appliquées - IMA
Lausanne, Switzerland
jorge.pena@unil.ch

Andres Upegui
Haute Ecole d'Ingénierie et Gestion du Canton Vaud - HEIG-VD
Reconfigurable and Embedded Digital Systems - REDS
Yverdon, Switzerland
andres.upegui@heig-vd.ch

Abstract

Self-adaptive autonomous hardware systems require on-chip heuristics to generate the circuit that constitutes the desired solution. In this paper, we present a population-oriented hardware architecture for particle swarm optimization with discrete recombination (PSO-DR), a hardware-friendly particle swarm that has shown to perform better than the standard PSO for certain parameter values and test functions. We present simulation and synthesis results showing the feasibility, performance, and advantages of the proposed architecture.

1 Introduction

Evolvable Hardware (EHW) allows four qualitative subdivisions according to its level of bio-inspiration: extrinsic, intrinsic, complete, and open-ended evolution [11]. *Extrinsic evolution* corresponds to evolutionary circuit design, as both fitness calculation and genetic operations are executed in software, and the resulting solution is implemented in hardware. In *intrinsic evolution*, a real circuit is used during the evolutionary process for fitness computation, even though most genetic operations are still executed in software. In *complete evolution*, there are systems in which all fitness evaluations, as well as genetic operations, are executed on hardware. Finally, *open-ended evolution* does not admit an externally imposed fitness criterion, but rather an implicit, emergent and dynamic one.

There are two distinct approaches to complete evolution: the *centralized* and the *population-oriented*. The *centralized* approach implies the use of a "genetic machine" implemented on the same evolving hardware substrate. An instance of this approach is Haddow and Tufte's pipelined hardwired genetic machine [14], where a single individual is multiplexed in time. Contrastingly, the distinctive feature of the *population-oriented* approach is the parallel implementation of the full population. This approach is used by Goeke et al. [3] in their implementation of a cellular automata, based on Sipper's *cellular programming* evolutionary algorithm [10], whose evolution takes place completely on-chip. In this algorithm, genetic operators are computed in a distributed way, as each automaton modifies its updating rule based on its own fitness and on that of its neighbors. However, cellular programming is not well suited for optimization problems. The algorithm is intended to find quasi-uniform cellular automata and thus converges to a population almost entirely composed of few distinct individuals.

EHW relies on the use of heuristic, stochastic, population-based optimization methods. The most common type of such methods is evolutionary algorithms (EA), while particle swarms [4] constitute the second major group. Particle swarms are starting to be used for EHW, specially in the context of extrinsic evolution [2, 6]. Intrinsic evolution of analog reconfigurable devices using particle swarms has also been proposed [12].

To the best of our knowledge, particle swarm optimization (PSO) has not yet been used to perform *complete* evolution of digital hardware systems. However, population-oriented hardware implementations of particle swarms have been proposed for solving different optimization problems, such as the inversion of large neural networks [9] and the dynamic adaptation of array antennas [5]. Particle swarms lend themselves well to population-oriented implementa-

Second NASA/ESA Conference on Adaptive Hardware and Systems(AHS 2007)
0-7695-2866-X/07 $25.00 © 2007 IEEE

tions, since their population sizes are often small (around 20 particles). This contrasts with typical GA population sizes, which could be one order of magnitude larger. Yet, the requirement of both multiplications and random numbers in the update formulas could be serious obstacles to the population-oriented approach. In [9], a solution to this problem is proposed by completely eliminating stochasticity in the update rules and restricting the values of the parameters to powers of two, so that multiplications could be realized by simple shift operations. Though this simplification worked well for the cited application, deterministic particle swarms performed worse than their stochastic counterparts. Thus, the solution could not be succesfully applied to other problems.

In this paper we present a population-oriented hardware implementation of the *particle swarm optimization with discrete recombination* algorithm (PSO-DR) [8]. Additionally, we validate the hardware usability of PSO-DR, proposing an optimized pipelined architecture that exploits the speed-up provided by the inherent parallelism of the hardware implementation, while still finding competitive solutions compared to software simulations.

The paper is organized as follows. Section 2 briefly introduces both particle swarm optimization and PSO-DR. A description of the proposed population-oriented hardware architecture is given in section 3; while simulation and synthesis results of its implementation are presented in section 4. Finally, section 5 concludes the paper and suggests some future work.

2 PSO-DR: Particle Swarm Optimization with Discrete Recombination

Particle Swarm Optimization (PSO) is a stochastic, population-based optimization method first introduced by Kennedy and Eberhart [4]. A particle swarm is defined as a group of M entities or particles, connected according to a given neighborhood topology, that explore an n-dimensional search space looking for the optimum of a function f. The search is performed iteratively, so that particles' states are updated from time step t to time step $t+1$ by the recursive application of a set of *update rules*. The state of the i-th particle at time step t is given by three vectors in the search space: its *position* ($\boldsymbol{x}_{i,t}$), its *velocity* ($\boldsymbol{v}_{i,t}$), and its *personal best* ($\boldsymbol{p}_{i,t}$).

The *i*-th particle of a *canonical particle swarm with inertia weight* modify its state according to:

$$\boldsymbol{p}_{i,t+1} = \arg\min \left\{ f\left(\boldsymbol{p}_{i,t}\right), f\left(\boldsymbol{x}_{i,t}\right) \right\} \tag{1}$$

$$\boldsymbol{v}_{i,t+1} = w \cdot \boldsymbol{v}_{i,t} + \mathbf{U}\left[0, \varphi_1\right] \otimes \left(\boldsymbol{p}_{i,t+1} - \boldsymbol{x}_{i,t}\right) + \mathbf{U}\left[0, \varphi_2\right] \otimes \left(\boldsymbol{p}_{b(i),t+1} - \boldsymbol{x}_{i,t}\right) \tag{2}$$

$$\boldsymbol{x}_{i,t+1} = \boldsymbol{x}_{i,t} + \Gamma\left(\boldsymbol{v}_{i,t+1}\right) , \tag{3}$$

where (a) $\Gamma(\cdot)$ is a component-wise clamping function that binds the velocity components within a given range, (b) $w \in [0,1]$ is the inertia weight, (c) $\mathbf{U}\left[lower, upper\right]$ is a vector of uniformly distributed random values between $lower$ and $upper$, (d) φ_1 and φ_2 are acceleration constants, (e) $\otimes$ is a point-wise vector multiplication operator, and (f) $\mathbf{p}_{b(i)}$ is the *neighborhood best*: the best *personal best* in the neighborhood of particle i.

Historically, two neighborhood topologies have been used the most extensively: Ring (or *lbest*) and All (or *gbest*). In Ring, particles are arranged as a one-dimensional cellular automata, each individual being connected to its immediate neighbors. In All, the topology is a fully connected one, the neighborhood of each particle being in this case the whole swarm. If an All topology is being used, the neighborhood best is the same for all the particles. It is called *global best* and represented by $\boldsymbol{p}_g$.

The canonical particle swarm with inertia weight presented in the previous lines is relatively well suited for population-oriented hardware implementations. Communication between particles is required only to calculate the neighborhood best. Apart from this issue, the state of each particle can be updated independently from each other. The velocity update rule (Eq. 2), however, still requires 2 random number generations and 3 multiplications per particle, per dimension. Hardware implementation of such operations is problematic since it could imply high silicon costs. In order to tackle this problem, we proposed particle swarm optimization with discrete recombination (PSO-DR) [8].

PSO-DR considers a swarm of particles arranged according to a Ring topology. A recombinant $\mathbf{r}_i$ is then defined for each particle i by performing a discrete recombination between the personal bests of its left and right neighbors. This recombinant is then used in lieu of the personal best or the neighborhood best in a deterministic velocity update rule. The *gbest* version of PSO-DR uses it in a velocity update rule of the form:

$$\boldsymbol{v}_{i,t+1} = w \cdot \boldsymbol{v}_{i,t} + \varphi_1 \cdot \left(\boldsymbol{r}_{i,t+1} - \boldsymbol{x}_{i,t}\right) + \varphi_2 \cdot \left(\boldsymbol{p}_{g,t+1} - \boldsymbol{x}_{i,t}\right) . \tag{4}$$

With the choices $\varphi_1 = \varphi_2 = 1$ and $w = 0.5$, that have shown good performance in different test functions [8], the algorithm involves no multiplications[1]. In addition to this, the random number generation requirements are minimal: only one bit per particle, per dimension, instead of two full

[1] Multiplication by 0.5 can be replaced by a trivial shift operation in fixed point representations.

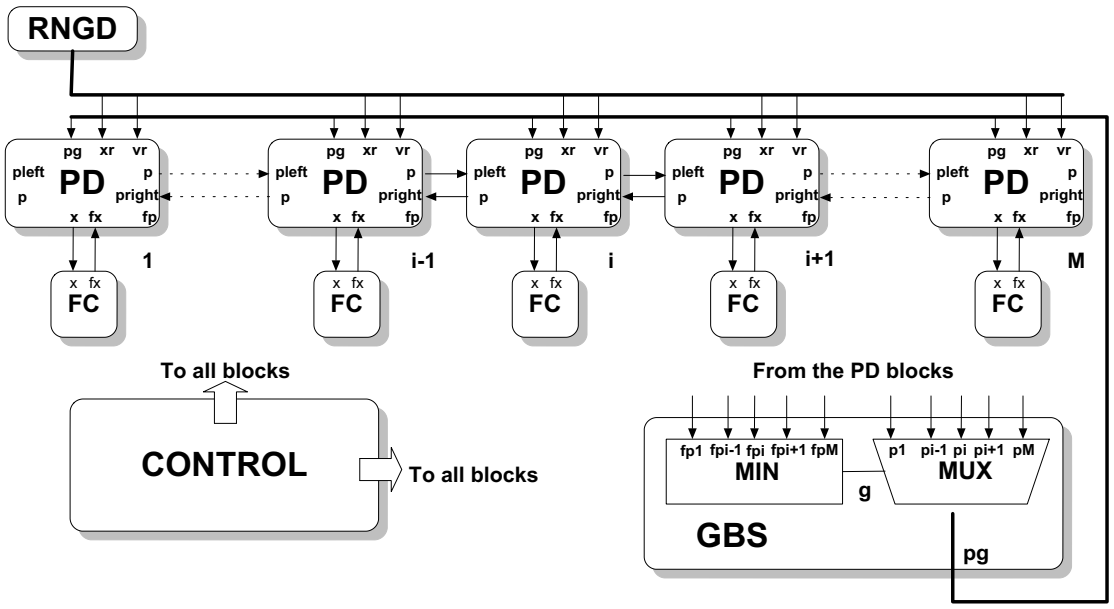

Figure 1. Proposed population-oriented architecture for PSO-DR. PD stands for particle datapath, FC for fitness computation, and GBS for global best selection

words, is required[2]. These two characteristics are ideal for having PSO-DR directly implemented in hardware, even following a population-oriented approach. For a better description of PSO-DR, see [8].

3 PSO-DR Architecture

A simplified block diagram of the whole system is shown in Fig. 1. It is comprised of M *particle datapath* blocks, interconnected according to a Ring topology, so that each *particle datapath* has access to the *personal bests* of its left and right neighbors (*pleft* and *pright* in the figure). Each *particle datapath* block implements a particle, stores the particle's state in memory and performs the required computation for updating it. Each *particle datapath* is associated with a *fitness computation* block, so that the whole population could be evaluated in parallel. In the case of a complete EHW application this *fitness computation* block would consist in the target digital circuit being evolved. Having a particle datapath and a *fitness computation* block for each particle assures the full population-oriented approach of the architecture: all the particles in the swarm are both evaluated and updated in parallel. Notice that the growth in area of this modular architecture is approximately linear versus the population size of the swarm.

[2]This random bit is needed to determine the recombinant target.

In addition to the *particle datapath* blocks, three *global* blocks are necessary for the circuit to work properly: (1) a *random number generation and distribution* (RNGD) block, that generates and distributes random bits to the array of particle datapaths, (2) a *global best selection* block, that receives the *personal bests* of the particle datapaths (and their fitness values), and computes the *global best*, broadcasting it to the particle datapaths, and (3) a *control* block for the synchronization of the whole operation.

3.1 Particle Datapath

A schematic of the particle datapath block is shown in Fig. 2. It consists of: (1) two dual-port RAMs with synchronous read (read through) for storing the multi-dimensional values corresponding to the position and the velocity of the particle (X-RAM and V-RAM), (2) a single-port RAM for storing the *personal best* of the particle (P-RAM), (3) a *personal best update* (P-*update*) block for controlling the updating of the particle best (Eq. 1), (4) a *velocity and position update* (VX-*update*) block for calculating the new velocity and position of the particle (Eq. 2 and 3), and (5) additional glue logic and registers.

In order to explain the operation of a *particle datapath* block, let us assume that the RAM blocks have already been properly initialized. Each line of RAM stores a single component or dimension of the respective vectors. First, fitness

0-7695-2866-X/07 $25.00

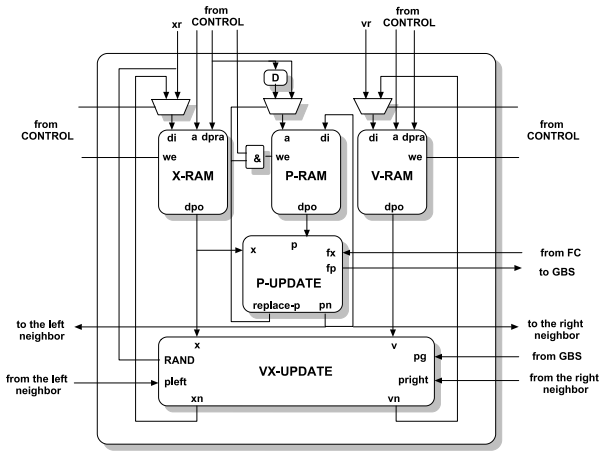

Figure 2. Particle datapath block. The clk and rst signals to the internal blocks are not drawn for the sake of clarity

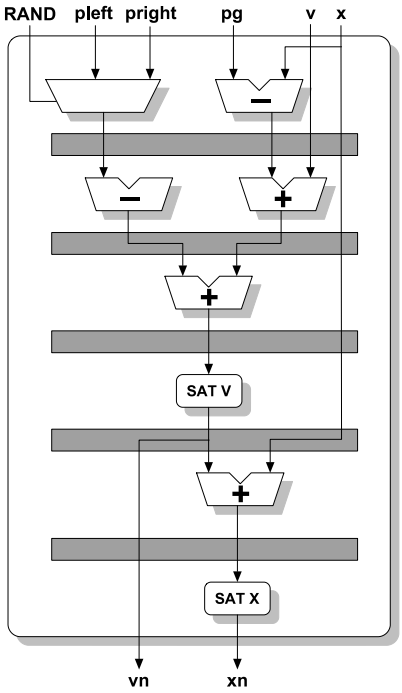

Figure 3. Update block. Gray bars represent registers between pipeline stages

of each particle is evaluated. The evaluation of the fitness of each particle is done by the *fitness computation* block, having access to the contents of the X-RAM memory. After fitness evaluation, the fitness of the current position (*fx*) is available for the P-*update* block. This block, which stores the fitness value of the *personal best* (*fp*), compares both fitness values. If *fx* is lower than *fp*, the *personal best p* is replaced by *x*. Otherwise the old *personal best* is kept. After this, the global best selection mechanism is activated: the *global best selection* block, that receives the fitness values of *personal bests* of all particles, calculates the global best (*pg*).

Finally, the VX-*update* block computes the new velocity and the new position of the particle, according to Eq. 2 and Eq. 3. The VX-*update* block is shown in Fig. 3. It is a pipelined datapath consisting of one multiplexer (for performing the discrete recombination process), two subtractors, three adders and two saturation blocks (one for the velocity and one the position). This pipelined architecture allows an efficient computation of the updated variables, by allowing the full computation, in a single clock cycle, of each dimension's update for both the velocity and the position of the particles.

3.2 Fitness Computation

The *fitness computation* block implements the function to be optimized by the PSODR algorithm. Several types of implementations are allowed for such block, depending on the target application of the optimization platform. In the specific case of evolvable digital hardware, this *fitness computation* block would consist of a set of *functional blocks* with some degree of reconfigurability at function specification or interconnectionism (or both). The functionality of these *functional blocks* can range from basic 2-inputs logic gates to more complicated functions such as arithmetic operators, neurons, or filters. What is really important is the possibility of this set of *functional blocks* to be reconfigured by means of a bit chain, in order to allow the evolutionary algorithm to find the final configuration.

One can identify two main implementations of this configurability when evolving circuits: real and virtual configuration. The main difference between them is the fact that in real configuration what is being evolved is the actual configuration bitstream of the reconfigurable device supporting the circuit, while in virtual configuration the configurable platform is a virtual configurable substrate implemented on top of the real one. Whether the reconfiguration is real or virtual depends on the technological constraints imposed by the reconfigurable device at hand and, in principle, it is transparent for the optimization algorithm execution. The fact of using real or virtual reconfiguration may only affect the system performance in two ways: by allowing a faster or slower evaluation of the individual, and by allowing a more compact implementation of the *fitness computation* supporting thus a higher number of particles.

In the specific case of this paper, we focus on the methodological issues that concern the hardware implementation of the optimization machine, and we do not focus on

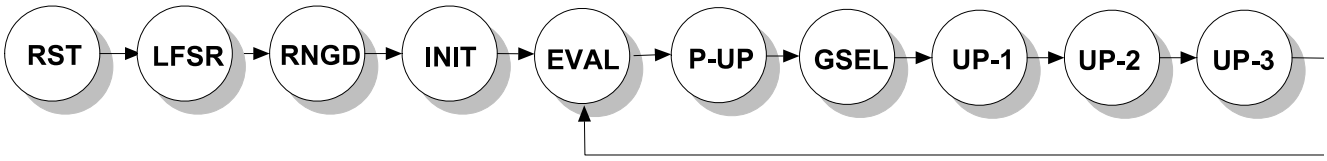

Figure 4. States and state transitions of the FSM of the control block

evolving a given digital circuit. Because of this, we used two mathematical test functions (described in section 4) as testbench for comparing the performance of our hardware-oriented algorithm implementation to software algorithms.

3.3 Random Number Generator and Distributor (RNGD)

The function of the RNGD module is to generate and distribute random numbers to the *particle datapath* blocks. It is comprised of a 16-bit LFSR expanded with a shift register with parallel output. Two random words (*xr* and *vr* in Fig. 1) are drawn in parallel from this stream and used for initialization of positions and velocities. During updating according to the PSO-DR updating rules, the only required random bit per particle and dimension (*RAND* in Fig. 2 and 3) is taken from the rightmost bit of the first of the two random words.

3.4 Global Best Selection

The *global best selection* block is a multiplexer-based switch block, that connects the *pg* input of each particle datapath block in the population with the output of the RAM (be it X-RAM or P-RAM) where the current global best is currently stored. The block is comprised of a set of pipelined minimum blocks arranged in a tournament-like manner, so that only the minimum values of a given stage are allowed to propagate. Comparison signals are collected and properly propagated so that at the end the index of the particle having the global best is found. Thanks to its architecture, the global selection block is able to calculate the index of the global best in just $\lfloor \log_2 M \rfloor$ clock cycles.

3.5 Control block

The *control* block is comprised of a finite state machine (FSM) augmented with a counter and decision logic for deciding state transitions, plus counters for generating addresses for the RAMs in the *particle datapath* blocks. A scheme of the states and the state transitions of the *control* block is shown in Fig. 4.

After an initialization phase (states RST, LFSR, RNGD and INIT), where particles' velocities and positions are randomly initialized, the fitness of the initial population is computed in the EVAL state. Then, during the P-UP and GSEL states, the P-*update* block and the *global best* selection block compute respectively the *personal best* and the *global best* values. Finally, velocities and positions are to be updated and their new values written in memory. Three states (UP-1, UP-2, and UP-3) are used for this. The first state fills the pipeline of the *update* block. When the pipeline is full it passes to state UP-2, and during UP-3 the pipeline flushes. In our specific implementation the pipeline is composed of 6 stages, but the same approach can be used for a larger number of stages.

After the UP-3 state, the system comes back to the EVAL state and all the process is repeated again from this point for performing a new iteration. The computation of a single iteration, consisting in the update of the state of all particles in the swarm can be achieved in $log_2 M + N + 7$ clock cycles.

4 System Setup and Results

The population-oriented hardware architecture presented in the previous section was described in VHDL for simulation and synthesis on an FPGA. The dimensionality of the search space (that defines the number of words of the position, velocity and *personal best* RAMs) as well as the number of bits in a data word (*DataWidth*) and the number of bits used to represent the fitness values (*FitnessWidth*), were left as parameters in the VHDL description. Parameters *DataWidth* and *FitnessWidth* determine the word size of the RAMs and of the registers in the architecture.

In a previous work [8], we used four benchmark mathematical functions (Sphere, Rosenbrock, Rastrigin and Griewank) to test the optimization capabilities of the PSO-DR algorithm. PSO-DR was found to be superior than a standard particle swarm algorithm, considering different population sizes and dimensionalities of the test functions.

Table 1. Number of bits m at the left and b at the right of the decimal point

Function	M	d	x_{min}	x_{max}
Sphere	8	24	-128	127.99609375
Rastrigin	5	27	-16	15.99951171875

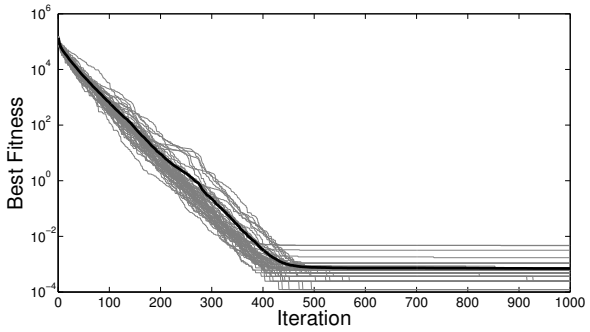

Figure 5. Learning performance according to the ModelSim simulation for the Sphere function. The , black curve represent the fitness of the global best averaged over 50 runs, while the gray narrow curves represent the fitness of the best solution for each single run

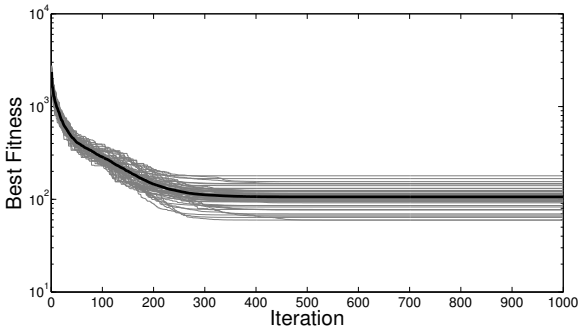

Figure 6. Learning performance according to the ModelSim simulation for the Rastrigin function. The same of conventions of Fig. 5 apply here.

The suitability of the algorithm being already shown, our interest was to verify the proper working of the designed architecture. In order to perform this verification, VHDL fixed-point models of two of the functions of the original benchmark (Sphere and Rastrigin) were coded with a dimensionality of $n = 32$, and values of $DataWidth = 16$ and $FitnessWidth = 32$ were used. The number of bits m at the left and d at the right of the decimal point used for the data words, and the resulting minimum and maximum values (x_{min}, x_{max}) of the search space of the two functions functions are referred in Table 1. A mathematical description of the Sphere and Rastrigin functions can be found in [8].

4.1 Simulation Results

For each function, 50 different runs were simulated, each one using a different 16-bit random seed for filling the LFSR during the LFSR state. The location of the minimum was kept fixed over time. A population size of $M = 16$ was used for both functions.

Fig. 5 and Fig. 6 show the learning performance for the two functions obtained directly from the ModelSim simulations. It can be observed how the system is able to optimize the functions. The first half of Fig. 5 show the characteristic straight line of a particle swarm minimizing the Sphere function, when the y-axis is in logarithmic scale. After the first 450 learning steps, there is no more progress in the minimization of the function, due to the finite precision of the fixed point representation.

In order to better assure the proper working of the designed hardware architecture, a fixed-point model of PSO-DR was coded in Matlab and run over the Sphere and Rastrigin functions. Fig. 7 and Fig. 8 show the comparison between the results obtained from the Matlab and the ModelSim simulations. Symmetric initialization of the positions and velocities of the particles was used for the Matlab models. Notice that the curves describing Matlab and ModelSim results are very similar. This leads to the conclusion of the good operation of the designed hardware. The little differences are due to the stochastic nature of the algorithms and the different used methods for obtaining random numbers in both models (extracting bit words in parallel from a shift register connected in series to a 16-bit LFSR for the ModelSim case, and using the *rand()* function in the Matlab case).

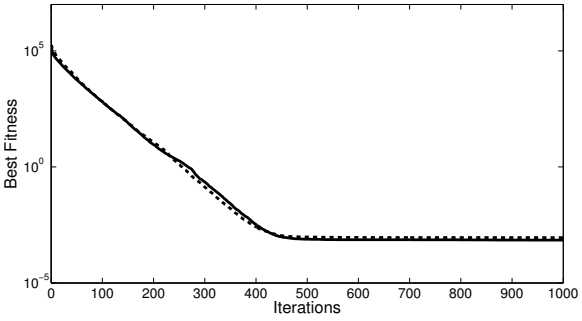

Figure 7. Learning performance according to the ModelSim (*solid line*) and fixed point Matlab (*dotted line*) simulations for the Sphere function

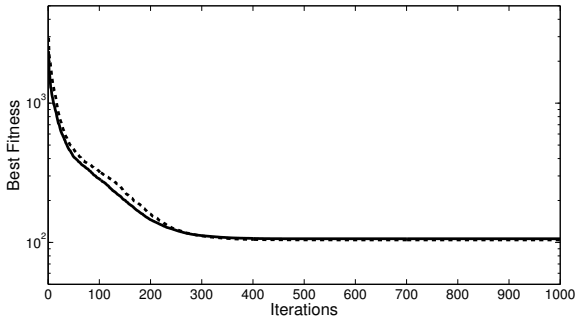

Figure 8. Learning performance according to the ModelSim (*solid line*) and fixed point Matlab (*dotted line*) simulations for the Rastrigin function

4.2 Synthesis Results

The proposed PSO-DR population-oriented architecture was synthesized on a commercial FPGA in order to determine the suitability of a real hardware implementation. We used a Virtex-4 XC4VLX25 from Xilinx, which has a maximum capacity of implementing 24192 logic gates. This FPGA has 96×28 configurable logic blocks (CLBs), with 4 slices each CLB, for a total number of 10752 slices. In addition to that, it has 72 18-Kb block RAMs, a maximum of 168 Kb distributed RAM and 48 XtremeDSP Slices (each one containing one 18×18 multiplier, an adder, and an accumulator).

The device utilization summary (results of the synthesis process in terms of slices, slice flip flops, look up tables and RAM blocks utilization) are shown in Table 2. Synthesis was done using the XST synthesis tool from Xilinx. The XC4VLX25 is not the largest device from the Virtex-4 LX FPGA family. The XC4VLX200, for instance, offers more than 8 times more logic resources and more than 4 times more RAM blocks. It can be seen, however, that more or less the 70% of the logic resources, as well as the 44% of the memory resources of the FPGA and the totality of the XtremeDSP slices are still free to be used in whatever hardware needs to be optimized.

According to synthesis reports, a clock of a minimum period of 6.214 ns (a maximum frequency of 160.919 MHz) can be expected for the design. With a population of $M = 16$ particles and a search space of dimensionality $n = 32$; the updating of the state of the whole swarm would take only $log_2 M + N + 7 = 43$ clock cycles, equivalent to 267.2 ns. The updating of the same swarm over a problem of the same dimensionality running in Matlab on a Intel Pentium M running at 1.6 GHz with 1GB of RAM, takes around 2.3 μs. This is equivalent to an approximate speedup of 8600 times of the hardware over the software implementation. Obviously, more efficient software implementations could be achieved, e.g., in C or assembler code, and these numbers are only valid as a very rough approximation to the expected speedup of the proposed hardware over a software solution.

Table 2. Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization
Slices	3454	10752	32%
Slice Flip Flops	4951	21504	23 %
4-input LUTs	5757	21504	26%
FIFO16/RAMB16s	48	72	66%

5 Conclusions and Future Work

This article has presented a population-oriented hardware implementation of a particle swarm optimizer. It features a RAM-based genotype-phenotype mapping, distributed and parallel processing capabilities, distributed storage, pipelined datapaths and fixed-point arithmetic. In addition to this, it is based on the hardware-friendly PSO-DR algorithm, and thus multipliers are needed.

The architecture was described in VHDL and successfully validated with two well established benchmark functions (Sphere and Rastrigin). The correct functionality of the proposed hardware was thus verified. Furthermore, the design is already synthesizable on an FPGA. It was shown that, when using a Xilinx Virtex-4, there are still enough resources for the adaptive circuit to be optimized, meaning that effective particle swarm-based, population-oriented, complete evolvable hardware is possible.

The current architecture supports static optimization problems only. Obviously, the importance of complete evolution EHW is its ability to achieve truly adaptive hardware, i.e., a dynamic optimization problem. Future work must thus target the inclusion of a mechanism for dynamic optimization in the proposed architecture. Extensions for dynamic optimization such as the use of *resetting* [1], or *forgetting and velocity resetting* [7] could be easily introduced, implying only minor changes and no important increase in the complexity of the design. A detection mechanism would also be necessary to track important changes in the environment so that the response method (resetting or forgetting and velocity resetting of particles) is triggered. Optionally, the response mechanism could be triggered on a periodic basis.

Another interesting feature to be included is the adaptability of the swarm size. The number of particles could be automatically and dynamically adapted according to the task's requirements. Thus, a big population could be used when facing difficult requirements, while a small population could be used otherwise. This would allow a more efficient utilization of the silicon resources by reusing the hardware substrate. This feature requires special self-reconfiguration mechanisms allowing a particle to self-replicate. Such mechanisms are not present in commercial FPGAs, but they are being currently targeted in custom bio-inspired devices such as the *ubichip* [15, 13].

References

[1] A. Carlisle and G. Dozier. Adapting particle swarm optimization to dynamic environments. In *International Conference on Artificial Intelligence*, volume I, pages 429–434, 2000, Las Vegas, NV.

[2] C. A. C. Coello, E. H. Luna, and A. H. Aguirre. Use of particle swarm optimization to design combinational logic circuits. In *ICES*, pages 398–409, 2003.

[3] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. Online autonomous evolware. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1259, pages 96–106. Springer-Verlag, 1997.

[4] J. Kennedy, R. Eberhart, and Y. Shi. *Swarm intelligence*. Morgan Kaufmann, 2001.

[5] G. Kokai, T. Christ, and H. H. Fruhauf. Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas. In *Proc. First NASA/ESA Conference on Adaptive Hardware and Systems*, pages 51–58, 2006.

[6] P. Moore and G. K. Venayagamoorthy. Evolving combinational logic circuits using a hybrid quantum evolution and particle swarm inspired algorithm. In *Evolvable Hardware*, pages 97–102, 2005.

[7] J. Peña. On-line, on-chip particle swarm optimization for adaptive hardware. Master's thesis, Advanced Learning and Research Institute, ALaRI. University of Lugano, 2006.

[8] J. Peña, A. Upegui, and E. Sanchez. Particle swarm optimization with discrete recombination: An online optimizer for evolvable hardware. In *Proc. First NASA/ESA Conference on Adaptive Hardware and Systems*, pages 163–170, 2006.

[9] R. Reynolds, R. Duren, M. Trumbo, and R. Marks. FPGA implementation of particle swarm optimization for inversion of large neural networks. In *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE (2005)*, pages 389–392.

[10] M. Sipper. *Evolution of parallel cellular machines the cellular programming approach*. Springer, Berlin, 1997.

[11] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, and A. Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97, 1997.

[12] P. Tawdross, S. K. Lakshmanan, and A. Konig. Intrinsic evolution of predictable behavior evolvable hardware in dynamic environment. In *HIS '06: Proceedings of the Sixth International Conference on Hybrid Intelligent Systems*, page 60, 2006.

[13] Y. Thoma, A. Upegui, A. Perez-Uribe, and E. Sanchez. Self-replication mechanism by means of self-reconfiguration. In *Workshop Procedings of the International Conference on Architecture of Computing Systems 2007 (ARCS'07)*. VDE Verlag, Berlin, 2007.

[14] G. Tufte and P. Haddow. Prototyping a GA pipeline for complete hardware evolution. pages 18–25, 1999.

[15] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. Moreno, and J. Madrenas. The Perplexus bio-inspired reconfigurable circuit. In *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, 2007.

0-7695-2866-X/07 $25.00