SOPC-Based Architecture for Discrete Particle Swarm Optimization

Amin Farmahini-Farahani, Sied Mehdi Fakhraie, Saeed Safari
School of Electrical and Computer Engineering, University of Tehran
a.farmahini@ece.ut.ac.ir, {fakhraie, saeed}@ut.ac.ir

***Abstract*—The complexity of modern chips is rising and fundamental changes in system design are necessary. System-on-a-Programmable-Chip (SOPC) concept is bringing a major revolution in the design of integrated circuits due to the flexibility it provides and the complexity it caters to. Particle Swarm Optimization (PSO) is a powerful function optimizer that is successfully used to solve problems in numerous fields. The main downside of PSO is that it has significant computation time because of sequential execution of software implementations. In this paper, an SOPC-based PSO framework is proposed. By implementing a hardware/software co-design of PSO, most of the computations can simultaneously be performed using hardware to reduce the computation time, while keeping the flexibility of software. The results indicate a speed-up of up to 100 times in the elapsed computation time.**

I. INTRODUCTION

In the last few years, Field Programmable Logic Device (FPLD) manufacturers have used programmable logic as a medium to develop System-on-a-Programmable-Chip (SOPC). By combining logic, memory and configurable processor core, embedded processor Field-Programmable Gate Array (FPGA) solutions allow system designers to integrate an entire system on a single device. SOPC platforms are commonly employed due to their ease of implementation and highly customizable nature. There are FPGA devices which offer hard or soft integrated processor subsystems. These devices have the flexibility to integrate memory, peripherals and other intellectual property (IP) for SOPC designs.

Particle Swarm Optimization (PSO) [1] is one of the evolutionary computation techniques based on swarm intelligence. Like the other evolutionary computation techniques, PSO is a stochastic population-based search algorithm. In PSO, a potential solution, called a particle, represents a point in the search space which has a fitness value and a velocity. Each particle flies through the solution space of problem and adjust its flying velocity to search for the global optimum according to its own and social historical experiences. Many researchers are interested in this algorithm and it has been investigated from various perspectives [2]. However, PSO process as other heuristic algorithms is time consuming. For many real-world applications, PSO can run for days, even when it is executed on a high performance workstation. To reduce the execution time of heuristic algorithms, several methods have been offered, including parallel and/or distributed processing of these algorithms along with its hardware implementation [3].

Although different hardware approaches for genetic algorithm have been proposed [4], a few hardware implementations of PSO have been reported in the literature. Kokai *et al.* [5] have employed hardware implementation of PSO in FPGA for the employment with blind adaptation of the directional characteristic of array antennas. They have introduced multi-swarm architecture in which each single swarm optimizes only a single parameter of the application. However, they have not reported the hardware cost, achieved frequency, and also performance.

Reynolds *et al.* [6] have implemented a modified particle swarm optimizer and neural network in FPGA. In their architecture, many of the computations are performed in parallel to reduce computation time compared to software implementation. They have employed two Xilinx XC2V6000 FPGAs. One FPGA was used for fitness function calculations, and the other was used for the particle swarm operations. At one hundred megahertz, their implementation was about sixty times faster than software implementation. Also, they have not reported the hardware cost.

In this paper, we propose an SOPC-based asynchronous discrete PSO system to speed up its processing time. The proposed system consists of two sections: fitness evaluation in software and all other PSO elements in hardware. By using a software implementation for the fitness function we achieve a system in which the potential increasing complexity of the fitness function does not alter the structure of the system.

The remaining sections are organized as follows: Section II gives an overview of PSO and its components. The proposed system is described in Section III. Section IV describes the SOPC implementation and explains the experimental results. Section V concludes the paper.

II. PARTICLE SWARM OPTIMIZATION

A. Particle Swarm Optimizer Algorithm

The particle swarm optimizer is a swarm intelligence algorithm that emulates a flock searching over the solution landscape by sampling points and converging the swarm on

1-4244-1378-8/07/$25.00 ©2007 IEEE.

the most promising regions. A particle moves through the solution space along a trajectory defined by its velocity. A particle consists of five components. The first component, *x*, is a vector that contains the current location in the solution space. *fitness* is the quality of the solution represented by the vector *x*, as computed by a problem-specific evaluation function. *v* is a vector that contains the velocity for each vector of *x*. The velocity of an *x* vector is the amount value the corresponding *x* value will change in the next iteration. $fitness_p$ is the fitness value of the best solution yet encountered by a particular particle. Each particle keeps track of its coordinates in the problem space, which are associated with the best solution it has achieved so far. Finally, *pbest* is a copy of *x* for the location of the best solution yet encountered by a particular particle.

Each particle is also aware of *gbest*, the particle reporting the current best fitness in the neighborhood for any given iteration, and $fitness_g$ which is the fitness value of *gbest*. Moreover, the entire swarm may be considered a single neighborhood, and *gbest* applies globally.

The heart of PSO algorithm is the process by which *v* is modified, forcing the particles to search through the most promising areas of the solution space again and again. At each iteration, the previous values of *v* constitute the particle momentum. PSO concept consists of velocity changes of each particle toward its *pbest* and *gbest* locations. Acceleration is weighted by a random term, with separate random numbers being generated for acceleration toward *pbest* and *gbest* locations. This randomness insures that the step size will be varying to avoid aliasing, where the particle endlessly follows the exact same path. The modified velocity and location of each individual particle can be calculated using the current velocity and the distance from $pbest_{id}$ and $gbest_d$, as shown in the following equations.

$$v_{id}^{k+1} = w.v_{id}^{k} + c_1 r_1 (pbest_{id} - x_{id}^{k}) + c_2 r_2 (gbest_d - x_{id}^{k}) \quad (1)$$

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \quad (2)$$

where x_{id}^{k} is the current location vector of the particle *i* at the iteration *k*, which has v_{id}^{k} as the velocity vector. This velocity vector satisfies $V_{\min} \le v_{id}^{k} \le V_{\max}$. Besides, there are five parameters that should be defined, *w* is the inertia weight factor, c_1 and c_2 are acceleration constants and r_1 and r_2 are uniform random numbers between 0 and 1.

B. Asynchronous Particle Swarm Optimization

The main difference between synchronous and asynchronous PSO concepts is the method used to update particle locations and velocities. Synchronous PSO updates particles locations and velocities at the end of every iteration. In contrast, Asynchronous PSO updates particle locations and velocities continuously based on currently available information. Synchronous PSO requires a synchronization point (calculation of all fitness values) before continuing to the next iteration.

C. Discrete Particle Swarm Algorithm

The original PSO algorithm can only optimize problems in which the elements of the solution are continuous real numbers. Nevertheless, many practical engineering problems are formulated as discrete optimization problems. Discrete particle swarm optimization [7] can be obtained by replacing (2) with (3).

$$\begin{aligned} &\textit{if } (rand < S(v_{id}^{k+1})) \quad \textit{then} \quad x_{id}^{k+1} = 1; \\ &\qquad\qquad\qquad\quad \textit{else} \quad x_{id}^{k+1} = 0; \end{aligned} \quad (3),$$

where *S(v)* is a sigmoid limiting transformation function $S(v) = 1 / (1 + e^{-v})$, and *rand* is a random number selected from a uniform distribution in [0.0, 1.0].

III. SOPC-BASED PSO FRAMEWORK

In this section, an asynchronous version of discrete PSO system has been proposed. We have implemented discrete PSO, since it is more agreeable to a hardware implementation despite its higher difficulty in software implementation due to demands of more random numbers. Hardware implementation of asynchronous PSO eliminates the demands of synchronization and multiple port memories. Also, the system utilizes hardware logics efficiently, and area cost is reduced.

In order to save area on the chip, some modifications to PSO are made. The simplest modification of the update equations is to use powers of two for the bias constants. This way, instead of using a slower and larger multiplier, a simple right shift is used. For this particle swarm implementation, the *pbest* and *gbest* bias constant was set to 1/64. This is a right shift of six bits.

SOPC-based PSO framework consists of two major segments. One is software implementation of the fitness function using an embedded processor, and another is hardware implementation of PSO operations. The proposed system and interconnections of different blocks are depicted in Fig. 1. Important parameters of the design are the bit-width for particle location and the bit-width for the particle velocity. These parameters are specified in the synthesis time. Increasing the bit-width of particles or increasing the number of particles does not increase the run-time of this hardware-based algorithm. The only limiting factor is the available area on the chip.

As shown in Fig. 1, there are four separate memories, two particle update blocks, random number generator (RNG) blocks, a determination block and an embedded processor and its program memory. Particle location memory and particle velocity memory store values of location vector (*x*) and values of velocity vector (*v*) for each particle, respectively. Pbest fitness value memory and pbest location memory hold fitness value of *pbest*, and values of the best location (*pbest* vector) for each particle, respectively. Two particle update blocks are employed to process particle location and velocity continually. Gbest register holds the yet achieved solution of problem (*gbest* vector).

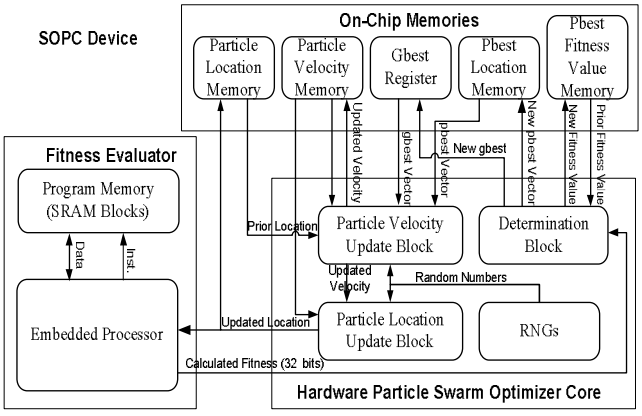

Figure 1. The proposed SOPC-based PSO architecture.

Here different blocks of the SOPC-based PSO architecture are described. Particle location update block feeds the embedded processor with updated location vectors. The embedded processor calculates the fitness of particles using software codes. Fitness evaluation is completely problem dependent and we have implemented it in software. Thus, the system can easily be adapted to solve different kinds of problems by loading new fitness functions into the program memory. The size of program memory depends on the software code utilization.

Determination block compares the calculated fitness of current location with the stored old ones to update the fitness value, Pbest memories, and Gbest register in every calculation step.

Particle velocity update block implements Equation (1) using hardware. Generating random numbers and multiplying numbers takes many clock cycles in every processor. These two operations take only two clock cycles in parallel implementation. At each step, updated particle location and velocity vectors are stored in particle location memory and particle velocity memory.

Random Number Generator blocks have a prominent role in PSO hardware implementation. For instance, for 32-bit particles and one thousand particle updates, 96,000 random numbers are needed (excluding random numbers required in fitness evaluation). Generally, linear feedback shift registers (LFSR) or cellular automata are used to generate random numbers [8]. The hardware random numbers produced by LFSRs deliver enough performance in order to use in evolutionary computation [8]. Therefore, we have used LFSRs for generating random data.

Initially, control unit sets the contents of particle location memory and particle velocity memory to random numbers using RNGs. Also, this unit generates control signals for all other blocks. This unit is composed of a state machine to track the operations of the algorithm. For clarity, this unit has been omitted from Fig. 1.

Particle location update block calculates new particle location according to Equation (3) in two clock cycles. The calculated results are passed to particle location memory and embedded processor. Particle location vector is indicated by a comparison between generated random number and velocity vector of that location. Several instances of this block are inserted in parallel to update particle location simultaneously.

IV. Hardware Implementation and Experimental Results

In order to measure the SOPC-based system performance, we have simulated and implemented this system on an Altera® Stratix 1S10ES Development Kit (chip EP1S10F780C6ES). All additional blocks are developed using Verilog hardware description language.

We have demonstrated the effectiveness of our SOPC implementation of PSO on three test functions. Experiments are performed to compare the different features of the SOPC-based PSO such as performance with the features of the software implementation of PSO. The utilized test functions are the OneMax function and two f_1 and f_2 functions of the standard DeJong Test Suite [9] with binary encoding. The algebraic forms of these functions are given in Table I. The first column represents the functions, and the second column shows the value range of particles in each function. The number of bits for each particle is shown in the third column.

To compare the performance of asynchronous PSO and synchronous PSO, the convergence rates for these test problems have been evaluated using software. In all of the experiments, the PSO algorithms use parameter values w=1, c_1=c_2=2, and the swarm size (number of particles) is 10. As shown in Table II, for DeJong f_1 problem, synchronous PSO outperforms asynchronous PSO. Asynchronous PSO performs slightly better than synchronous PSO for other problems. Asynchronous PSO is well suited for the hardware implementation due to its ability of continually updating particles, whereas synchronous PSO entails a synchronization point before advancing the next iteration. The synchronization point causes some blocks to stand idly by in hardware implementations, considerably increasing elapsed time.

For our system, one Nios II/f [10] is employed to evaluate fitness values. The maximum operating frequency of the system for all the problems is almost 50 MHz. In all the problems, PSO process terminates after 100 iterations.

Table III summarizes memory bits and logic cells needed for different implementations of asynchronous discrete PSO. In the SOPC-based implementation, software program of the test function is prepared with C language and it is run on the processor. Memory bits of the software part are related to software codes of the fitness function loaded into the program memory of the Nios processor, and the number of memory bits of the hardware part is determined by the required memory bits to hold PSO vectors and values. The used logic cells consist of the hardware area of the Nios processor and PSO specific blocks. In software implementation, C program codes of complete asynchronous

PSO algorithm besides the fitness function are loaded into the program memory of the Nios processor. The number of used logic cells determines the Nios processor consumed area.

TABLE I. TEST SUITE

Function	Search Range	Dimension
$OneMax(\bar{x}) = \sum_{i=1}^{32} x_i, \quad n = 32$	{0, 1}	32
$f_1(\bar{x}) = \sum_{i=1}^{n} x_i^2, \quad n = 3$	[−5.12, 5.12)	30
$f_2(\bar{x}) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$	[−2.048, 2.048)	24

TABLE II. ACHIEVED RESULTS FOR TEST FUNCTIONS. VALUES ARE THE AVERAGE FITNESS VALUES OF 100 TRIALS AND 100 ITERATIONS (LESS IS BETTER)

PSO Algorithm	OneMax	DeJong f_1	DeJong f_2
Synchronous	0.194	0.389	6.627
Asynchronous	0.168	0.491	5.299

Table IV shows the average processing times for different asynchronous PSO implementations. The second column shows the elapsed time of the software implementation of PSO written in MATLAB and implemented on a single 3-GHz Pentium IV processor, 1-GB RAM, and Windows XP Pro OS. The third and fourth columns represent the elapsed times of the software implementation of PSO on Nios processor and SOPC-based implementation of PSO, respectively. In all of the implementations, DeJong problems take more clock cycles than OneMax problem, because fitness function of OneMax problem is simple. The major bottleneck of the SOPC-based system is fitness evaluation in Nios II processor. Although it is possible to design specific hardware for fitness evaluation (instead of software implementation) for simple functions to speed up the system, modularity and flexibility are lost and also this makes the design more cumbersome.

According to Table IV, in the SOPC-based implementation, evolution of 100 iterations of DeJong f_2 problem at 50 MHz is just under 210 ms, i.e., more than 5000 fitness evaluations are made every second. The improvement ratio of SOPC-based PSO over software PSO on Nios processor is up to 100 times and that over the implementation on Pentium processor is up to 11 times.

TABLE III. USED MEMORY BITS AND LOGIC CELLS IN ASYNCHRONOUS DISCRETE PSO IMPLEMENTATIONS

Problem	Implementation Type	Memory Bits	LCs
OneMax	SOPC-based	52K + 4.2K (SW + HW)	6213
	Software on Nios	360K	2850
DeJong f_1	SOPC-based	160K + 3.9K (SW + HW)	5842
	Software on Nios	448K	2850
DeJong f_2	SOPC-based	168K + 3.2K (SW + HW)	4787
	Software on Nios	448K	2850

TABLE IV. ELAPSED TIMES OF SOFTWARE AND SOPC-BASED REALIZATIONS OF ASYNCHRONOUS DISCRETE PSO

Problem	Software PSO on Pentium IV	Software PSO on Nios (50 MHz)	SOPC-based PSO (50 MHz)
OneMax	70 ms	1042 ms	10.1 ms
DeJong f_1	2333 ms	9302 ms	212 ms
DeJong f_2	1737 ms	8562 ms	209 ms

PSO highly depends on random numbers, and one reason for the achieved improvement is that generating random numbers using a software program requires many clock cycles. Moreover, software implementations cannot take advantage of the inherent parallelism of PSO algorithm since instructions are executed serially. While in hardware, most computations of particles are performed in parallel.

V. CONCLUSION

In this paper, we have proposed an embedded implementation for discrete PSO using System-on-a-Programmable-Chip (SOPC) concept. With respect to the optimization function, the calculation time for each iteration is distinct. Thus, this architecture can be used in real-time PSO applications. The proposed realization operates up to 100 times faster than its corresponding software implementation. The experimental results confirm the benefits of our mixed solution in comparison to having an exclusively software or hardware implemented system. Our solution provides a tradeoff between the speed of a full-hardware solution and the flexibility of a pure software implementation.

REFERENCES

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," *in Proc. IEEE Intl. Conf. Neural Networks*, vol. 4, 1995, pp. 1942-1948.

[2] X. Hu, Y. Shi, and R. Eberhart, "Recent advances in particle swarm," in *Proc. IEEE Cong. Evolutionary Computation*, 2004, pp. 90-97.

[3] Z. Konfrst, "Parallel genetic algorithms: Advances, computing trends, applications and perspectives," in *Proc. IEEE Parallel and Distributed Processing Symp.*, Apr. 2004, pp. 162-169.

[4] Z. Zhu, D. J. Mulvaney, and V. Chouliaras, "A novel genetic algorithm designed for hardware implementation," *Intl. J. of Computational Intelligence*, vol. 3, no. 4, pp. 281-288, 2006.

[5] G. Kokai, T. Christ, and H. H. Frhauf, "Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas," in *Proc. NASA/ESA Conf. Adaptive Hardware and Systems*, 2006, pp. 51-58.

[6] P. Reynolds, R. Duren, M. Trumbo, and R. Marks II, "FPGA implementation of particle swarm optimization for inversion of large neural networks," in *Proc. IEEE Swarm Intelligence Symp.*, June 2005, pp. 389-392.

[7] J. Kennedy and R. Eberhart, "A discrete binary version of the particle swarm algorithm," in *Proc. IEEE Conf. on Syst., Man, and Cybernetics*, 1997, pp. 4104-4109.

[8] P. Martin, "An Analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C," in *Proc. Genetic and Evolutionary Computation Conf.*, July 2002, pp. 837-844.

[9] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, U. Michigan, Ann Arbor, 1975.

[10] *Nios II Processor Reference Handbook*, Altera Corp., May 2006.