The Open Artificial Intelligence Journal, 2010, *4*, 20-29

Open Access

An Efficient Distributed Genetic Algorithm Architecture for Vector Quantizer Design

Wen-Jyi Hwang[1], Chien-Min Ou*[,2], Peng-Chieh Hung[1], Cheng-Yen Yang[1] and Tun-Hao Yu[1]

[1]*Department of Computer Science and Information Engineering, National Taiwan Normal University, Taipei, Taiwan, 117, R.O.C:* [2]*Department of Electronic Engineering, Ching Yun University, Taoyuan, Taiwan, 320, R.O.C*

Abstract: This paper presents a novel distributed genetic algorithm (GA) architecture for the design of vector quantizers. The design is based on a multi-core architecture, where each island of the GA is associated with a hardware accelerator and a softcore processor for independent genetic evolutions. An on-chip RAM with a mutex circuit is adopted for the migration of genetic strings among different islands. This allows a simple and flexible migration for the implementation of hardware distributed GA. Experimental results shows that the proposed architecture has significantly lower computational time as compared with its software counterparts running on multicore processors with multithreading for GA-based optimization.

Keywords: Distributed GA, SOPC, Multi-core system, Vector quantizers.

1. INTRODUCTION

Genetic algorithms (GAs) [1] are a class of general-purpose search algorithms for solving optimization problems by simulating natural evolution over populations of candidate solutions. The algorithms have been found to be effective for solving problems in engineering, science and business. However, when they are applied to complex problems, the computational complexity may become very high.

One way for reducing the computational time is to employ the distributed GA algorithm [2-4]. There are multiple populations in a distributed GA. Each population evolves independently most of the time. Different populations may exchange genetic strings occasionally. With smaller population size, the distributed GA may be able to converge at faster rate while finding good solutions. To find near optimal solutions, however, large population size may still be desired. This will still result in long computation time.

The objective of this paper is to present a VLSI architecture for distributed GA. The architecture is able to accelerate the GA even for large population size. The application considered in this paper is vector quantization (VQ) [5]. When applied for VQ codeword training, the GA requires large storage size and long training time. Therefore, the VQ design is a good example for verifying the effectiveness of the proposed GA architecture.

In the proposed architecture, each island of the GA is associated with a hardware accelerator and a processor for independent genetic evolutions. An on-chip RAM with a mutex circuit is adopted for the migration of genetic strings among different islands. This allows a simple and flexible migration for the implementation of hardware distributed GA.

Although some existing hardware architectures [6, 7] can be used for the design of the hardware accelerator for genetic evolution in each island, these architectures have the following drawbacks. First of all, large storage size is required for processing the genetic strings. Usually two set of population memories are used for the regeneration process. One memory contains the parent strings; the other stores the child strings after the regeneration. In addition, there is overhead for switching one memory to another at the beginning of a new generation. The second drawback is that the regeneration process is based on the fitness function. The selection of parents therefore may need large chip area for hardware implementation. In addition, the mutation and crossover operations also result in high area cost when concurrent processing over all the genetic strings is desired.

The proposed hardware accelerator is able to eliminate the drawbacks stated above. The steady-state GA [8, 9] is used for the accelerator so that regeneration, mutation and crossover operations can be simplified. The accelerator consists of population memory unit, mutation and crossover unit, fitness evaluation unit, and survival test and update unit. It contains only one population memory for reducing the area cost. Both the mutation and crossover operations are performed concurrently for accelerating the GA. In addition, a pipeline architecture with direct memory access (DMA) operation is adopted for the fitness function evaluation. A hardware sorting structure is adopted for survival test.

The proposed architecture has been implemented on field programmable gate array (FPGA) devices [10] so that the processors in the architecture can be implemented by softcore CPUs [11]. Using the reconfigurable hardware, we are

*Address correspondence to this author at the Department of Electronic Engineering, Ching Yun University, Chungli Taiwan, 320, R.O.C;
Tel: 886-3-458-1196-5121; Fax: 886-2-458-8924;
E-mail: cmou@cyu.edu.tw

then able to construct a system on programmable chip (SOPC) system for the genetic VQ design. As compared with its software counterparts running on multicore processors with multithreading, numerical results reveal that the proposed FPGA-based GA architecture attains higher performance with significantly lower training time for VQ design. These fact demonstrate the effectiveness of our design.

2. STEADY-STATE GA FOR VQ DESIGN

Before presenting the architecture, we first briefly review the steady-state GA [8] for VQ design. The goal of a VQ for data clustering is to partition a large data set $\chi = \{x_1,...,x_t\}$ into N non-overlapping clusters $C_1,...,C_N$, where $N >> t$. The partitioning process is based on a set of codewords $\{y_1,...y_N\}$, where the codewords and the vectors in χ are of the same dimension *w*. Given a vector $x \in \chi$, the x will be assigned to the cluster C_i when

$$i=a(x)=\arg \min_{1 \le j \le N} d(x,y_j), \quad (1)$$

where $d(u,v)$ denotes a distance measure between two vectors *u* and *v*. In this paper, the squared distance is adopted as the distance measure. When applied for data reduction applications such as data compression, a vector *x* will be represented by the codeword y_i when $i = \alpha(x)$. One cost function for the data reduction is the average distortion for representing *x* by y_i, as shown below

$$D = \frac{1}{wt} \sum_{i=1}^{t} d(x_i, y_{\alpha(x_i)}). \quad (2)$$

Given a data set χ, the objective of the VQ design is to find a set of codewords $\{y_1,...y_N\}$ minimizing D in eq.(2).

In the steady-state GA for VQ design, there are *P* genetic strings for the genetic operations. Each string *r* represents a set of *N* codewords $\{y_1,...y_N\}_r$. Note that these strings are strings of vectors, not strings of binary numbers.

There is no concept of generation in steady-state GA. Let population *S* be the set of *P* genetic strings, which are called the parent strings. Initially, the *P* strings in *S* are randomly generated. Two strings (denoted by r_1 and r_2) in *S* will be selected for mutation and crossover for creating a new child string (denoted by *c*). The fitness value of the child string is then evaluated and compared with the fitness value of all the parent strings in *S*. If the new string is inferior to all the parent strings in *S*, no parent string will be removed. Otherwise, the parent string with lowest fitness value is replaced by the child string.

Note that because each string for the VQ design is actually a codebook, the memory access time for string retrieval may be long. Consequently, the retrieval process for r_1 and r_2 may be time-consuming. To reduce the memory access time, in the algorithm, the previous r_1 becomes the new r_2 and then the new r_1 is chosen randomly from *S*. This selection scheme reduces the memory access time by half.

As the process of selection, crossover, mutation, and survival/replacement continues, the overall fitness of population will increase and the survival rate of new off-spring will diminish. At some point, the offspring survival rate will drop to zero. At this point, evolution has probably ceased and the algorithm may be terminated. The steady-state GA algorithm is more effective for the hardware design. Only one population memory is required. In addition, crossover and mutation operations only operate on r_1 and r_2 instead of all strings in the population memory.

In the distributed steady-state GA, there are *M* islands. Each island *i* is associated with a separated population S_i. Genetic strings in each S_i are evolved independently using steady-state GA until the offspring survival rate of S_i drops to zero. A migration process is then activated by importing *K* genetic strings from island *j*, $j \neq i$, and exporting *K* strings to island *k*, where *j* and *k* are randomly selected. After the migration, another new evolution based on steady-state GA is re-started. An iteration of the distributed steady-state GA consists of a migration process and a steady-state GA evolution. After each iteration, the best genetic string with highest fitness value is recorded. The iterations of the distributed GA is halted when identical best genetic string are found for *L* consecutive iterations. The entire distributed steady-state GA is completed when all the iterations of all the *M* islands are halted.

3. THE PROPOSED ARCHITECTURE

3.1. Architecture for Steady-State GA

We start with the architecture for steady-state GA, which is depicted in Fig. (**1**). It contains population memory, crossover & mutation unit, fitness evaluation unit, survival test & update unit, and Avalon bus interface. Both the population memory and crossover & mutation unit contain random number generators (RNGs). In this architecture, the population memory unit is devoted for storing the genetic strings. Moreover, the random selection of parent strings for subsequent crossover and mutation operations is also included here. This selection is based on the RNG inside the population memory unit. All the crossover and mutation operations are performed concurrently in the crossover & mutation unit for producing a new child string *c*. The fitness value of the resulting string is then evaluated by the fitness evaluation unit.

Based on the fitness value, the goal of the survival test & update unit is to determine whether the child string *c* will survive. If it is the case, the parent string in the population memory unit with the worst fitness value will be replaced by

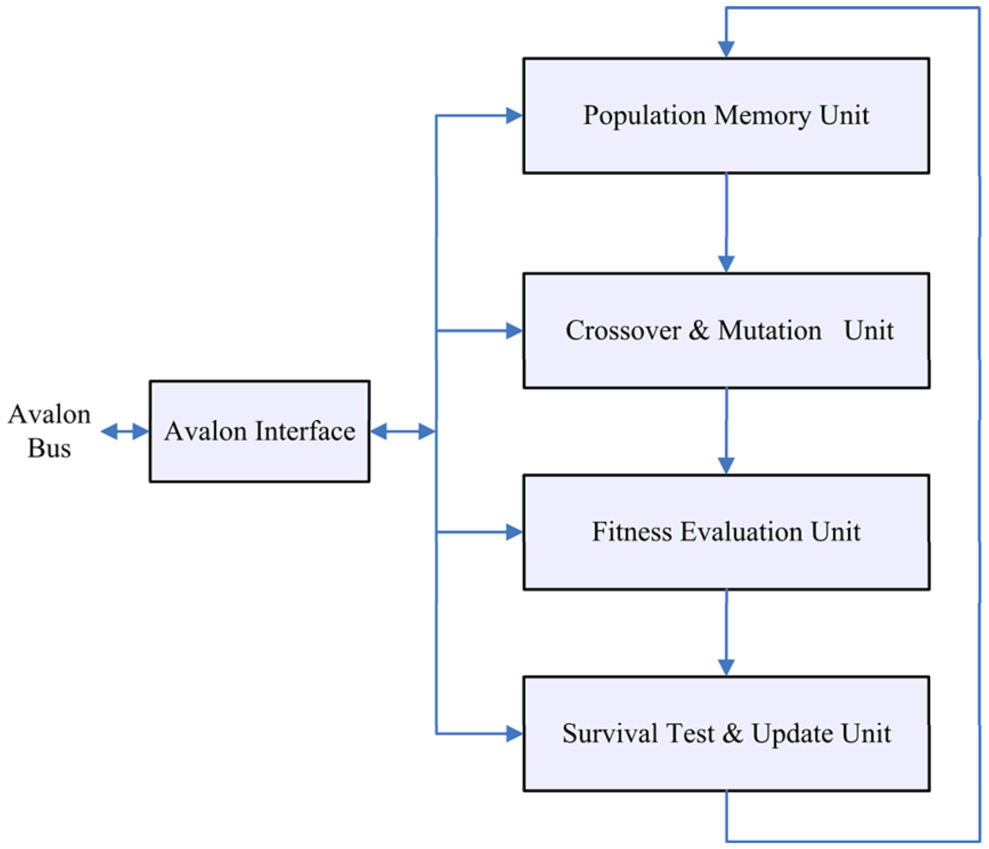

Fig. (1). The proposed hardware architecture of steady-state GA.

the child string. Each unit in Fig. (**1**) will be described in detailed as shown below.

Population Memory Unit

The population memory contains a 2-pot RAM and a RNG unit. The 2-port RAM contains *S*, the set of *P* genetic strings. In our design, the implementation of the RAM is based on the embedded memory, which is provided by some FPGA devices such as Altera Stratix II. The goal of RNG unit in the population memory unit is to select randomly a string r_1 for the subsequent crossover and mutation operations. In our design, the cellular automata (CA) is adopted for the VLSI implementation of random number generator [12] due to its simplicity and regularity of the design.

Mutation and Crossover Unit

Fig. (**2**) shows the basic structure of the mutation and crossover unit, which contains three shift registers for storing the strings r_1, r_2 and *c*, respectively. A number of RNGs, comparators, multiplexers and counters are then used for crossover and mutation. The major advantage of this architecture is that the crossover and mutation can be performed concurrently with low area cost.

As shown in Fig. (**2**), SHIFT REGISTER 1 and SHIFT REGISTER 2 contain strings r_1 and r_2, respectively. Note that the architecture does not randomly select new r_1 and r_2 from the population memory. In fact, only new r_1 is chosen from population memory. The new r_2 is actually the previous r_1. The memory access time and routing overhead can then be significantly reduced. Based on the algorithm, in the architecture, The SHIFT REGISTER 1 obtains r_1 from the population memory unit. The SHIFT REGISTER 2 obtains r_2 from SHIFT REGISTER 1.

The crossover operations are accomplished by concurrently shifting the strings in SHIFT REGISTER 1 and SHIFT REGISTER 2 to MUX 1. Each shift register will shift one codeword at a time. As shown in Fig. (**2**), MUX 1 is a switch selecting the codewords of either r_1 or r_2, and route them to SHIFT REGISTER 3, which contains the resulting child string *c*. The control line of MUX 1 is connected to a comparator, which compares the value of RNG 1 to that of a counter. The counter records the number of shifts made by the shift registers. The value of RNG 1 serves as a threshold here. When the counter value is less than the threshold, codewords of SHIFT REGISTER 1 (i.e., r_1) goes to SHIFT REGISTER 3. Otherwise, codewords of r_2 will be selected. Consequently, the value of RNG 1 determines the crossover point. The value will be randomly generated prior to the shifting operations.

We also observe from Fig. (**2**) that the output codeword of MUX 1 will pass through the mutation unit before arriving the SHIFT REGISTER 3. Fig. (**3**) shows the architecture

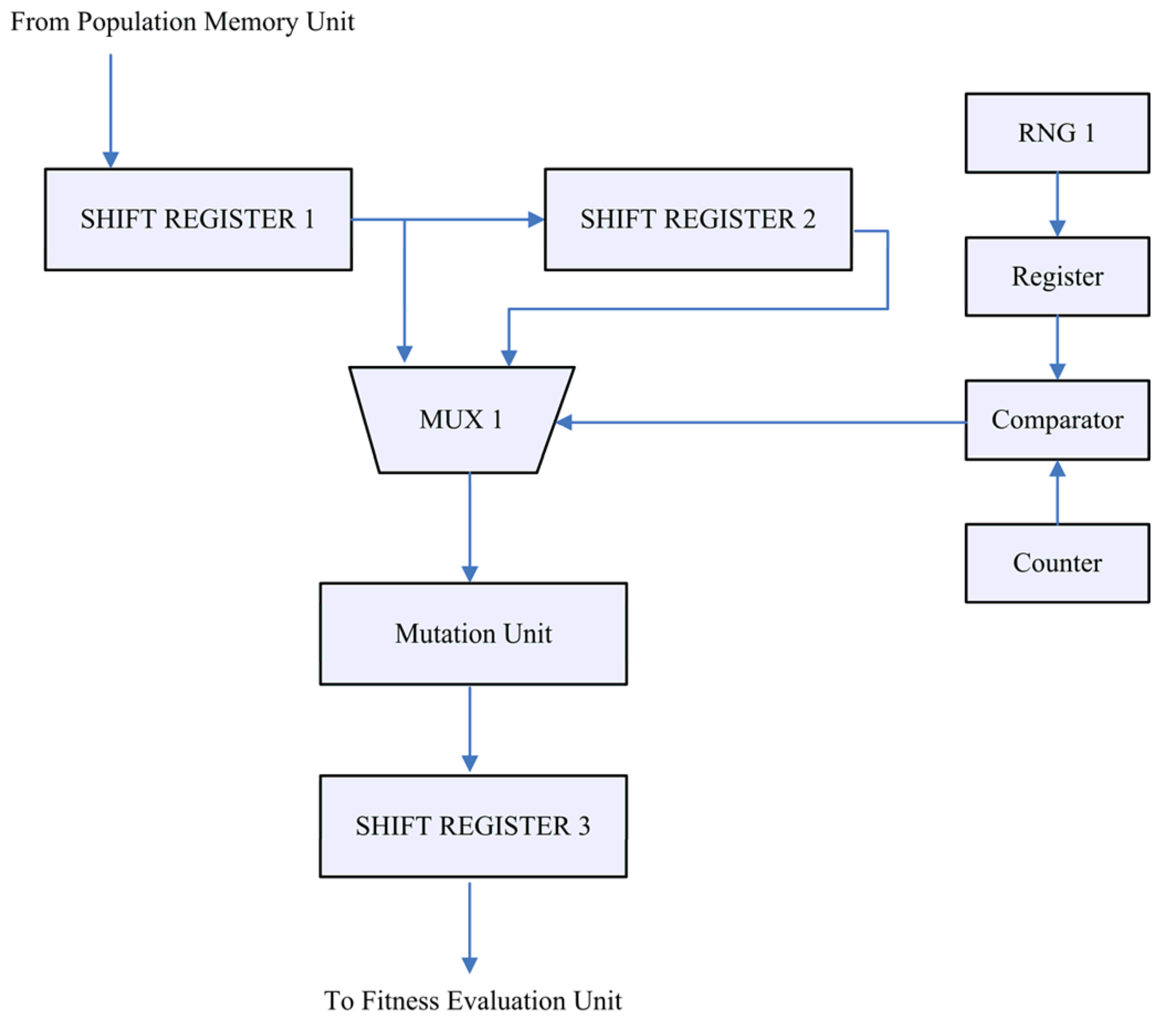

Fig. (2). The architecture of crossover and mutation unit.

of the mutation unit. As shown in the figure, all w components of the output codeword mutate concurrently. The mutation circuit for each component *i* consists of 2 RNGs (termed RNG *ia* and RNG *ib*), one register (termed register *i*), one comparator (termed comparator *i*), one multiplexer (termed mux *i*).

The probability for mutation P_b is stored in a separate register, and is broadcasted to all the mutation circuits. In the mutation circuit for each component *i*, the value of RNG *ia* is first compared with the P_b. The component *i* will be mutated when the value of RNG *ia* is less than P_b. The mutated value is then determined by RNG *ib*.

Fitness Evaluation Unit

The goal of the fitness evaluation unit is to compute the average distortion of the mutated child string stored in SHIFT REGISTER 3 using eq.(2). Fig. (**4**) shows the architecture of the fitness evaluation unit, which is an *N*-stage pipeline, where *N* is the number of codewords. The pipeline fetch one training vector $x \in \chi$ per clock from the input.

The *i*-th stage, $i = 1,\ldots,N$ of the pipeline compute the squared distance between the training vector at that stage and the *i*-th codeword of the child string stored in the SHIFT REGISTER 3 of the mutation and crossover unit shown in Fig. (**2**). The squared distance is then compared with the current minimum distance up to the *i*-th stage. If distance is smaller than the current minimum, then the *i*-th codeword becomes the new current optimal codeword, and the corresponding distance becomes the new current minimum distance. After the computation at the *N*-th stage is completed, the current optimal codeword and current minimum distance are the actual optimal codeword and the actual minimum distance, respectively.

Survival Test and Update Unit

This unit contains a hardware sorting circuit [13], which sorts the *N* parent strings in a descending order according to their fitness values. After the fitness evaluation operation is completed, the fitness value of the child string is used as the input to the sorting circuit. When the distortion of the string is larger than the parent string with lowest fitness value, the child string is not survival, and no updating operation is necessary. Otherwise, the parent string with highest distor-

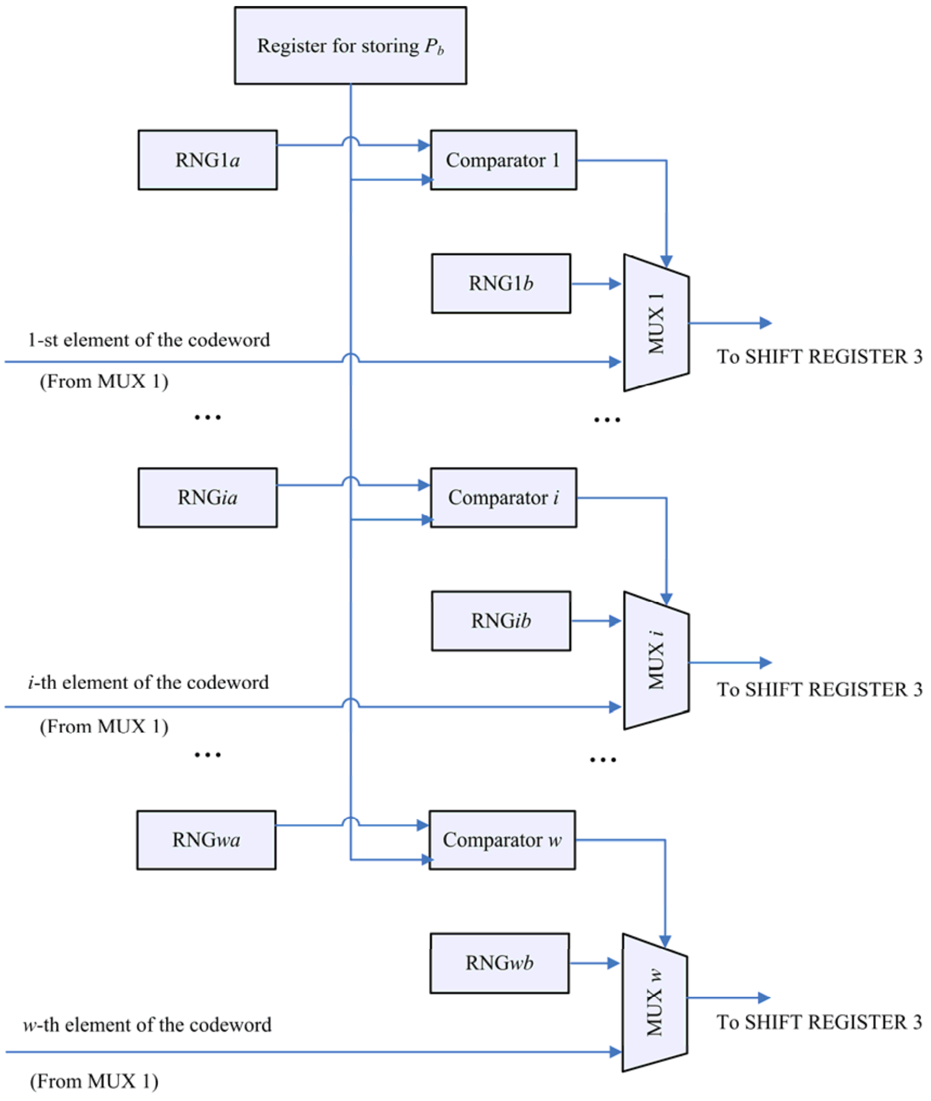

Fig. (3). The architecture of mutation unit.

tion is replaced by the child string. The sorting circuit is then activated to determine the new parent string with the highest distortion.

3.2. Architecture of Each Island

The architecture of each island of the proposed distributed GA architecture is depicted in Fig. (**5**). From the figure, it can be observed that each island is associated with one hardware accelerator, one direct memory access (DMA) controller, and one softcore NIOS II processor. The hardware accelerator is adopted for speeding up the steady state GA computation associated with each island. The architecture of the accelerator is shown in Fig. (**1**).

Recall that training vectors are required for fitness evaluation. The set of training vectors are stored in the main memory. The DMA controller shown in Fig. (**5**) is used for delivering the training vectors from the main memory to the input of the fitness evaluation unit shown in Fig. (**4**).

The softcore processor in Fig. (**5**) is used for coordinating the distributed steady-state GA operations in each island. The softcore processors in different islands will operate independently. Each processor triggers the hardware accelerator and DMA controller for regeneration, crossover and mutation, fitness evaluation and survival test and update operations of its own island. The processor then checks for the offspring survival rate. When the survival rate drops to zero, it will then record the best string in the population, and activates the migration process for genetic strings. The migration process consists of exchanging the K genetic strings between the island and an on-chip RAM outside the island. After the migration process, the accelerator and DMA controller will then be activated again for another evolution. The softcore processor will also determine when should the iterations be terminated. A software flowchart for the softcore processor associated with each island is shown in Fig. (**6**).

3.3. Architecture of the Distributed Steady-State GA

Fig. (**7**) depicts the architecture of the distributed steady-state GA. There are M islands, and each island has its own circuit. Therefore, there are M modules for genetic evolution in Fig. (**7**), where each module corresponds to one

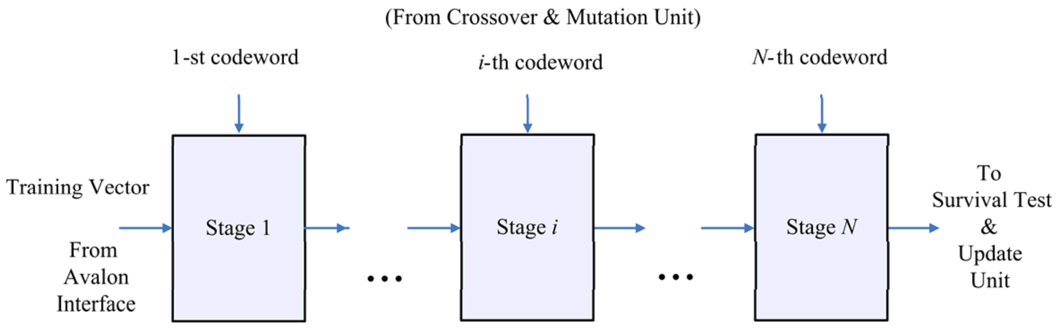

Fig. (4). The architecture of the fitness evaluation unit.

island. It has the architecture as shown in Fig. (**5**). Because each module has a softcore processor, the architecture in Fig. (**7**) can be viewed as an *M*-core circuit. It can also be observed from Fig. (**7**) that all the modules share the same main memory, which contains the set of training vectors for fitness evaluation.

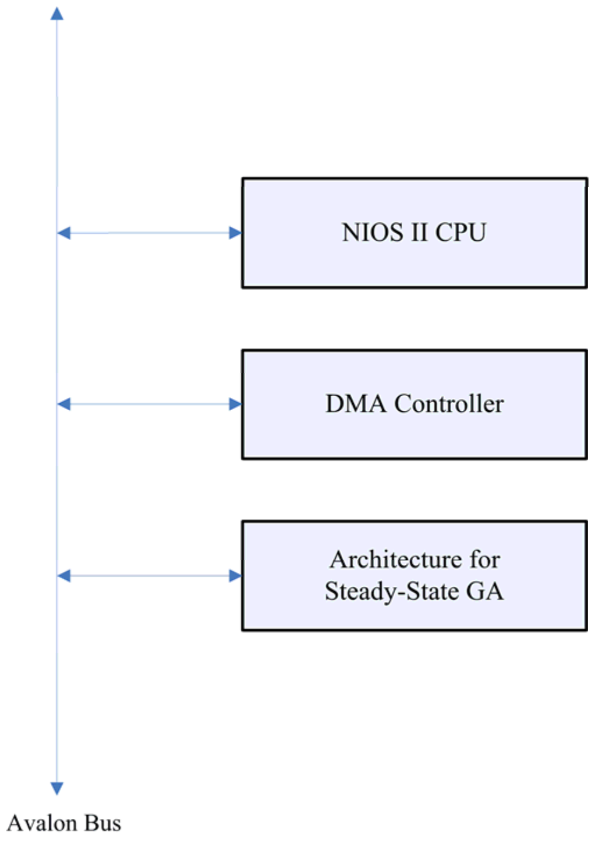

Fig. (5). The architecture of each island of the proposed distributed GA architecture.

One major difficulty for the hardware implementation of distributed GA algorithm is the string migration. Each island may randomly select a target island for string exchange. When the number of islands is large, the circuit for migration can be complicated. To simplify the string migration process, in addition to the *M* modules, the distributed GA architecture also contains a memory for the string migration. The memory, termed string migration cache, is shared by all the modules.

Based on the string migration cache, it is not necessary to design a circuit for selecting the target module for migration for each module. To perform the migration, each module competes for the exclusive access to the string migration memory. The winner exchanges *K* genetic strings with the cache. When the winner accomplishes its migration process, it releases the memory. At this point, another module may acquire the memory for the string migration.

The migration process based on the string migration memory does not actually perform the string exchange between two islands. In fact, an island winning the memory obtains the new string from its previous winner. However, the island does not donate its strings to its previous winner. The island will give its strings to the next winner. Therefore, in our architecture, there is no string exchange. Instead, the strings migrate from one winner to the next. The first winner will obtain new strings which are randomly generated in the memory. This scheme may have comparable performance to the string exchange scheme with significantly less hardware implementation complexity.

To protect the string migration memory from data corruption that can occur if more than one modules attempts to use the memory at the same time, a hardware mutex core is also used, as shown in the Fig. (**7**). The mutex allows cooperating modules to agree that one of them should be allowed mutually exclusive access to the string migration memory.

Note that, without the hardware mutex, the function for data corruption protection normally requires two separate "test" and "set" instructions between which, the processor in another module could also test for availability and succeed. This situation may leave two processors both thinking they successfully acquired mutually exclusive access to the string migration memory when clearly they did not. The atomic operation provided by the hardware mutex is essential for our string migration scheme.

4. EXPERIMENTAL RESULTS

This section presents some physical performance measurements of the proposed FPGA implementation. The target FPGA device for the hardware design is Altera Stratix II 2S60 [14], which contains 288 DSP blocks and 24176 adaptive logic modules (ALMs) [15]. The Altera Quartus II with

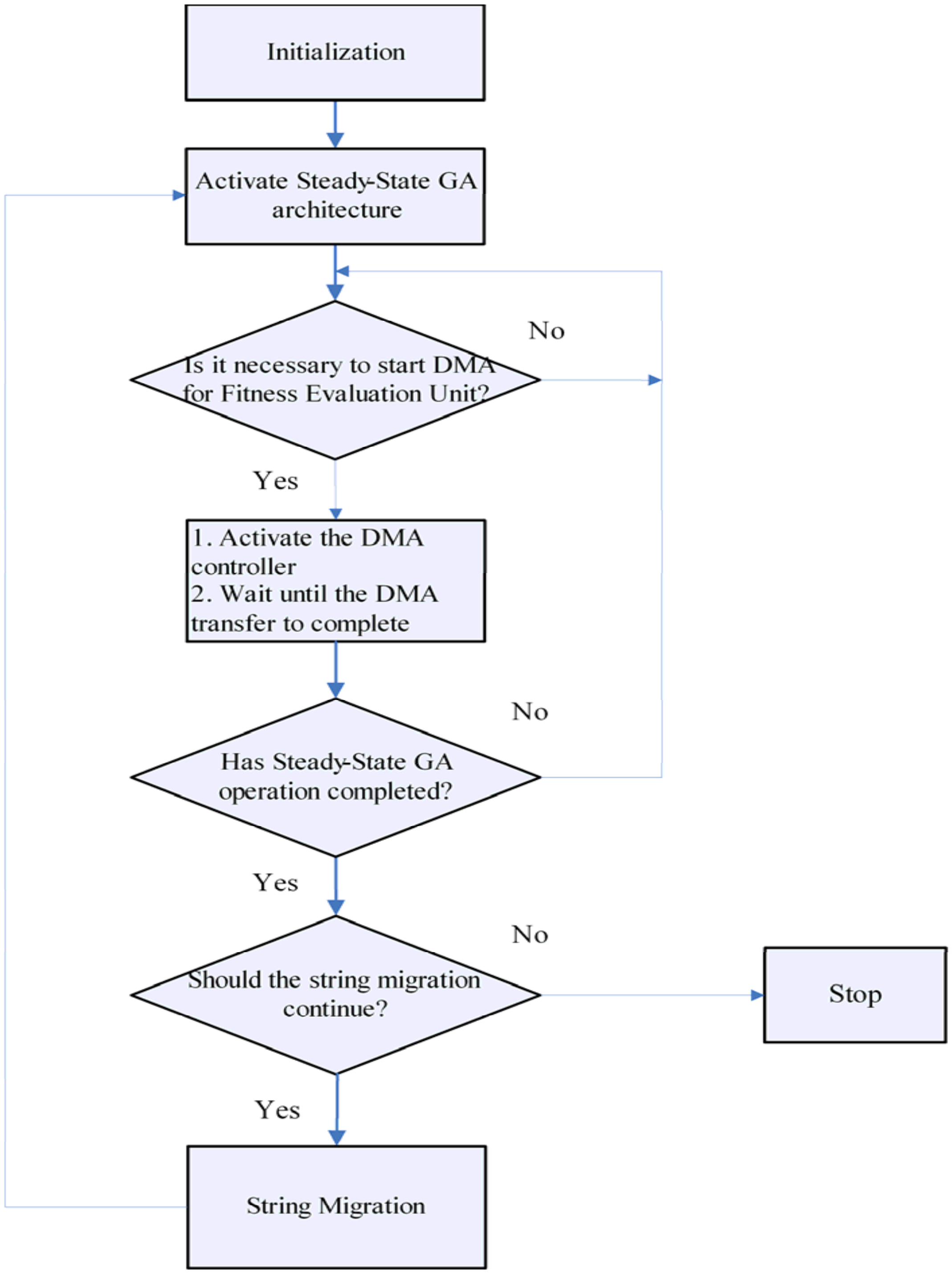
Initialization
Activate Steady-State GA architecture
Is it necessary to start DMA for Fitness Evaluation Unit?
No
Yes
1. Activate the DMA controller
2. Wait until the DMA transfer to complete
Has Steady-State GA operation completed?
No
Yes
Should the string migration continue?
No
Stop
Yes
String Migration

Fig. (6). The flowchart for the softcore processor.

SOPC builder is used as the platform for our design. The vector dimension of codewords is $w = 2\times 2$. There are 32 codewords (i.e., N = 32) in the VQ. The mutation probability is P_b = 0.03125. The number of islands for the distributed GA is 3 (i.e., M = 3).

Table **1** shows the area cost of the architecture of steady-state GA, the architecture of each island, and the architecture of distributed GA. The population size is P = 16 for each island. As revealed in the table, the steady-state GA architecture consumes only 3096 ALMs. The population memory of the architecture is implemented by the embedded memory of the FPGA. The consumption of the population memory bits for the steady-state GA circuit is 16384 bits. Moreover, the circuit also uses 128 digital signal processing (DSP) 9-bit elements of the FPGA device for the implementation of squared distance computation in the fitness evaluation unit.

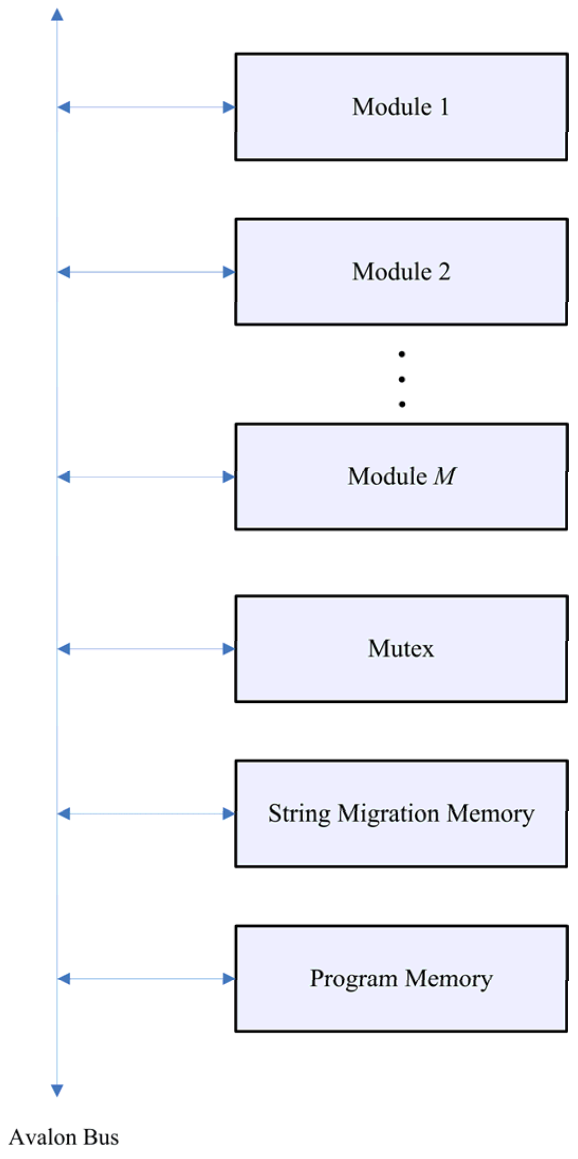

Fig. (7). The architecture of the distributed steady-state GA.

The NIOS softcore CPU [11] and DMA controller also consume hardware resources. Therefore, it can be found from Table **1** that the area cost of the island architecture is higher than that of the steady-state GA architecture. However, the island architecture uses 7162 ALMs, which is only 30% of the ALMs of the target FPGA device. The distributed GA system contains 3 island architectures, mutex circuit and SDRAM controller. Hence, the ALMs consumed by the entire distributed GA system is slightly higher than triple of that consumed by each island.

Fig. (**8**) shows the distribution of average distortion of the distributed GA with $M = 3$. The distribution is obtained by 200 independent executions of the distributed GA. The number of genetic strings of each island is $P = 16$. There are three islands. Therefore, the total number of genetic strings is 48. The training set contains 65536 vectors drawn from the image "Lena." The distribution of basic steady-state GA (i.e., $M = 1$) is also included in the figure for comparison purpose. The number of genetic strings is 48 for the steady-state GA. The two GAs therefore are compared on the basis of the same number of total genetic strings. From the figure, we see that both GAs have similar distributions. Therefore, the employment of the distributed GA does not degrade the performance for the VQ design.

The CPU time of various distributed GA implementations are compared in Table **2**. The distributed GAs are implemented with $M = 3$. We also set $P = 16$ for each island. The software implementations are developed on the 2.6GHz multicore workstation HP-ML570. As shown in Table **2**, both single threading and multi-threading schemes are adopted for the software implementation. The evolution of the 3 islands are executed sequentially by a single thread in the single threading implementation. By contrast, for the multi-threading implementation, the evolution of each island is processed by an independent thread. Different threads are executed by separate cores so that evolution of all the islands are processed in parallel. The number of training vectors is 65536 from the image "Lena".

It can be observed the Table **2** that the CPU time of the proposed architecture is significantly lower than that of its distributed GA software counterparts. In fact, the speedup of the proposed architecture over its single threading and multi-threading software counterparts are 65 and 30, respectively. The speedup is defined as the execution time (2.6-GHZ Pentium IV CPU) of the software implementation divided by the execution time (50-MHz NIOS softcore CPU) of the SOPC system using the proposed GA architecture as the custom user logic. Based on (Fig. **8** and Table **2**), we conclude that the proposed architecture is able to accelerate the genetic optimization process without sacrificing its performance.

Table **3** shows the area costs, average distortion and execution time of the proposed SOPC system for various population size P for each island. The distortion and execution time are obtained by averaging those of 200 independent

Table 1. The Area Cost of the Architecture of Steady-State GA, the Architecture of Each Island, and the Architecture of Distributed GA

	Steady-State GA Circuit	**Architecture of Each Island**	**Architecture of Distributed GA**
ALMs	3096(13%)	7162(30%)	24139(99%)
Block memory bits	16384(3%)	613120(24%)	643968(25%)
DSP block 9-bit elements	128(44%)	136(47%)	288(100%)

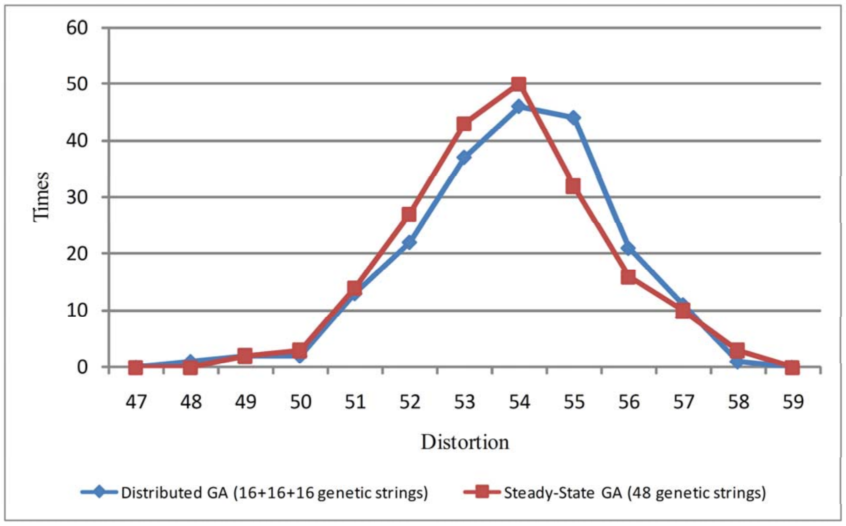

Fig. (8). The distribution of average distortion of the distributed GA.

Table 2. The CPU Time of Various Distributed GA Implementations

	Average Distortion	Average Execution Time
Software distributed GA (single-thread)	54.06	85208.7ms
Software distributed GA (multi-thread)	54.24	38555.9ms
Hardware distributed GA	54.11	1293.8ms

Table 3. The area Costs, Average Distortion and Execution Time of the Proposed SOPC System for Various Population Size *P* for Each Island

P	Entire System Embedded Memory Bits	Entire System ALMs	Average Distortion	Average Execution Time
4	619392(23%)	22145(92%)	58.24	869.5ms
8	627584(24%)	22829(94%)	57.41	991.2ms
12	635776(24%)	23493(97%)	55.53	1153.7ms
16	643968(25%)	24139(99%)	54.11	1293.8ms

executions. It can be observed from Table **3** that the area cost of the entire system becomes only slightly higher as *P* increases. The average execution time grows linearly with *P*. In addition, the average distortion can be effectively reduced as *P* becomes larger.

To further investigate the effectiveness of the proposed architecture, Fig. (**9**) shows the speedup of the proposed architecture over its multi-threading software counterpart for various numbers of training vectors from the 4 training images “Baboo”, “Hill”, “Bridge” and “Girl”. The population size is *P* = 16 for each island. It can be observed from Fig. (**9**) that the speedup increases with the training set size. This is because the training set is used for the fitness evaluation in the GA. The training set is often stored in the main memory, and therefore requires long memory access time. In addition, the inverse of the average distortion *D* in eq.(2) is used as the fitness function. The computational complexity therefore is high. In our architecture, the DMA and pipeline techniques are used for reducing the memory access time and the computational time for fitness evaluation. Therefore, our design has high speedup over the software implementation when the training set size is large. In particular, when the training set contains 65536 vectors, the execution time of our SOPC system is 3952.6 ms; whereas, the execution time of software system is 85017.5 ms. The speedup is 22. These facts demonstrates the effectiveness of our design.

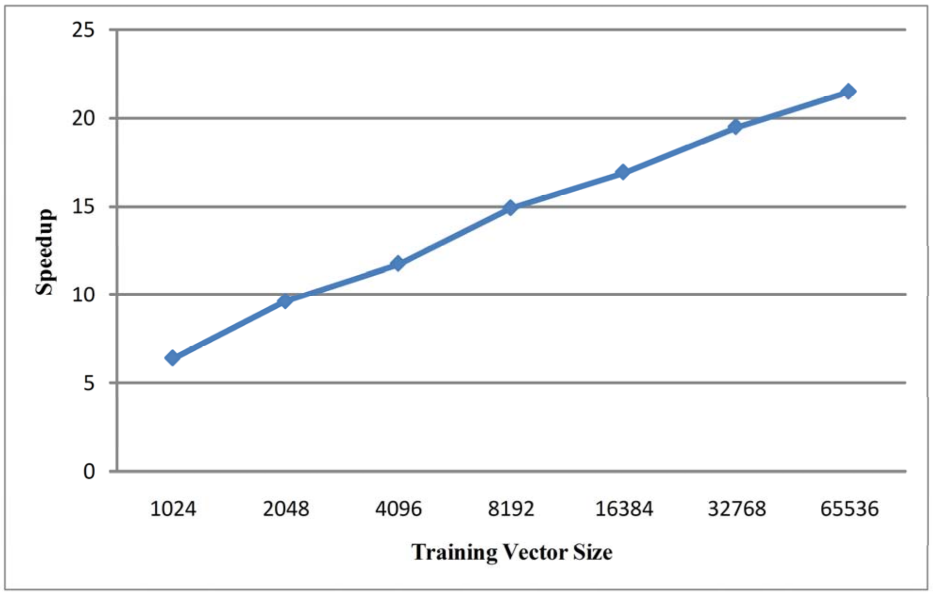

Fig. (9). The speedup of the proposed architecture over its multi-threading software counter-part for various numbers of training vectors from four training images.

CONCLUDING REMARKS

The proposed architecture has been found to be effective for the hardware implementation of distributed GA. The employment of steady-state GA for the evolution within each island is able to simplify the complexity for the design of hardware accelerator. By assigning each island an independent core and hardware accelerator for genetic evolution, the proposed architecture is able to accelerate the genetic optimization process without sacrificing its performance. The population size can be increased to lower the average distortion without the consumption of large hardware resources. The speedup of the architecture over its software counterpart also increases with the training set size. The proposed architecture is therefore an effective alternative for genetic optimization applications requiring realtime computation without sacrificing its performance.

REFERENCES

[1] A. E. Eiben and J. D. Smith, *Introduction to Evolutionary Computing*, Springer, 2003.

[2] E. Cantu-Paz, "*Efficient and Accurate Parallel Genetic Algorithms*", Norwell: Kluwer, 2000.

[3] A. Chipperfield and P. Fleming, "Parallel Genetic Algorithms", *Parallel and Distributed Computing Handbook*, A. Y. H. Zomaya, Ed., McGraw-Hill, New York, 1996, pp. 1118-1143.

[4] D. Whitley and T. Starkweather, "GENITOR II: A distributed genetic algorithm," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 2, pp. 189-214, 1990.

[5] A. Gersho and R.M. Gray, *Vector Quantization and Signal Compression*, Norwood: Kluwer, 1992.

[6] N. Nedjah and L. Mourelle, "Hardware architecture for genetic algorithms", *Lecture Notes in Computer Science*, vol. 3533, pp. 554-556, 2005.

[7] M. Tommiska and J. Vuori, "Implementation of genetic algorithms with programmable logic devices," *Proc. 2nd Nordic Workshop on Genetic Algorithms and Their Applications*, 1996, pp. 111-126.

[8] K. Rasheed and B.D. Davisson, "Effect of global parallelism on the behave of a steady state genetic algorithm for design optimization," *In Proceedings of the Congress on Evolutionary Computation*, Washington, DC, 1999.

[9] G. Syswerda, "A study of reproduction in generational and steady state genetic algorithms", *Foundations of Genetic Algorithms*, G. Rawlins, Ed. San Fransisco, Morgan Kaufmann, 1991, pp.94-101.

[10] S. Hauck and A. Dehon, *Reconfigurable Computing*: *The Theory and Practice of FPGA-Based Computing*, Morgan Kaufmann, Publishers, CA, 2008.

[11] *NIOS II Processor Reference Handbook*, 2008, Altera Corporation. http: //www.altera.com/ literature/ lit-nio2.jsp.

[12] P.D. Hortensius, R.D. McLeod, and H.C. Card, "Parallel random number generation for VLSI systems using cellular automata", *IEEE Transaction on Computer*, vol. 38, No. 10, pp. 1466-1473, 1989.

[13] W.J. Hwang, H.Y. Li, Y.J. Yeh, and K.F. Chan, "FPGA implementation of competitive learning with partial distance search in the wavelet domain," *Progress in Neurocomputing Research,* G. B. Kang, Ed., NOVA Science Publisher, 2008, pp. 203-221.

[14] *Stratix II Device Handbook*, 2008, Altera Corporation San Joe. http:// www.altera.com/ literature/ lit-nio2.jsp.

[15] M. Hutton, J. Schleicher, D. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bourgeault, A. Lee, H. Kim and R. Saini, "Improving FPGA performance and area using an adaptive logic module", *Lecture Notes in Computer Science*, vol. 3203, pp. 135-144, 2004.

Received: August 12, 2009 Revised: November 01, 2009 Accepted: November 03, 2009