High-Speed General Purpose Genetic Algorithm Processor

Seyed Pourya Hoseini Alinodehi, Sajjad Moshfe, Masoumeh Saber Zaeimian, Abdollah Khoei, and Khairollah Hadidi

Abstract—**In this paper, an ultrafast steady-state genetic algorithm processor (GAP) is presented. Due to the heavy computational load of genetic algorithms (GAs), they usually take a long time to find optimum solutions. Hardware implementation is a significant approach to overcome the problem by speeding up the GAs procedure. Hence, we designed a digital CMOS implementation of GA in 0.18 μm process. The proposed processor is not bounded to a specific application. Indeed, it is a general-purpose processor, which is capable of performing optimization in any possible application. Utilizing speed-boosting techniques, such as pipeline scheme, parallel coarse-grained processing, parallel fitness computation, parallel selection of parents, dual-population scheme, and support for pipelined fitness computation, the proposed processor significantly reduces the processing time. Furthermore, by relying on a built-in discard operator the proposed hardware may be used in constrained problems that are very common in control applications. In the proposed design, a large search space is achievable through the bit string length extension of individuals in the genetic population by connecting the 32-bit GAPs. In addition, the proposed processor supports parallel processing, in which the GAs procedure can be run on several connected processors simultaneously.**

Index Terms—**Genetic algorithm processor (GAP), dual-population, hardware genetic algorithm, multi-GAP, parallel processing.**

I. Introduction

GENETIC algorithm (GA) is one of the most famous evolutionary algorithms, which is inspired from natural evolution of living beings. It finds solutions through the search in problem space. This mechanism enables the GA to solve difficult and nonlinear problems that are not easily solved via classic methods. On the other hand, search in problem space, i.e., generation of possible solutions and evaluating them, requires considerable time. The time factor imposes limitations on the application of GA in control and problems with heavy computations. Since running the GA on a dedicated circuit is usually faster than running it on the software, hardware implementation is a way to compensate the GAs inherent computational complexity. However, the need for circuit implementation of GA also exists when using a computer is not possible or there is strict requirement for using a dedicated hardware.

A. Related Works and Motivations

Thanks to the advantages of implementing GA on the hardware, many researchers have proposed various genetic algorithm processors (GAPs). In [1], a general-purpose GA chip, which is able to tune crossover and mutation rates, is proposed. In [2], another general-purpose GA with adjustability in some of its parameters was implemented on a 0.5 μm process chip. An implementation of steady-state GA that supports some simple types of parallel processing was introduced in [3]. To the same design, the means for dynamic adjustment of migration frequency was added in [4]. In [5], an intellectual property (IP) for GA, which its parameters are set by software before fabrication, was described. In [5], a lookup table can be used instead of external fitness units (FUs). One of the main disadvantages of this model of implementation is that there is no way to modify the parameters of GA after fabrication. In [6]–[13], GA is implemented on the field-programmable gate arrays (FPGAs). The proposed hardware in [7] and [13] operate as coprocessors for the software. In [14], an FPGA implementation of GA is proposed for circuit partitioning in very large-scale integration physical design automation. Another FPGA implementation of GA is introduced in [15], which is capable to work with pipelined fitness functions. Also, Fernando *et al.* [16] reported a soft IP core for GA that was tested on the FPGA. An FPGA-implemented GA was described in [17] with a neural network as its fitness function. Some kinds of modified GAs (compact GA) on FPGAs were introduced in [18]–[20]. Some variants of evolutionary algorithms intended for hardware implementation, such as compact GA and differential evolution [21], differ from the proposed steady-state GA algorithmically. As mentioned, several previous works are realized on FPGAs that make parameter changing more convenient, but reduces their efficiency and speed in contrast to application-specific integrated circuits (ASICs). Among previous ASIC implementations of GA, many have limited flexibility in parameter adjustment or totally are application-specific.

S. P. Hoseini Alinodehi was with the Microelectronics Research Laboratory, Urmia University, Urmia 57159, Iran (e-mail: sph.alinodehi@gmail.com).

A. Khoei and K. Hadidi are with the Microelectronics Research Laboratory, Urmia University, Urmia 57159, Iran (e-mail: a.khoei@urmia.ac.ir; kh.hadidi@urmia.ac.ir).

S. Moshfe is with the Department of Electrical Engineering, Islamic Azad University, Arsanjan Branch, Arsanjan 7376153161, Iran (e-mail: s.moshfe@iaua.ac.ir).

M. Saber Zaeimian was with Islamic Azad University, Lahijan Branch, Lahijan 1616, Iran (e-mail: m.zaeimian@gmail.com).

The primary goal of the proposed hardware GA, like many other implementations, is to accelerate the processing of GA. With respect to the results of previous works, ASIC implementation was chosen for speed considerations. Besides, it was aimed that the proposed hardware to be highly configurable.

B. Overview of the Proposed Work

In this paper, a new generic steady-state GAP is proposed. Generic in the sense that it is not restricted to a special problem and is capable to perform optimizations in all applications. For this purpose, fitness evaluation of possible solutions (chromosomes) takes place out of the processor in external systems designed just for a particular application. The proposed GAP only performs genetic operations, thus sends out the chromosomes to be evaluated in external FUs and then receives their fitness. External FUs can be an ASIC, FPGA, microcontroller, or even a computer program.

The proposed GAP is able to work with the maximum of 32 bits of data. This means that the output data (output chromosomes) and their respective fitness can be up to 32 bits. Nonetheless, several GAPs can serially connect to each other and act as a processor with higher bit support. In this way, a set of n GAPs transfer chromosomes with $32 \times n$ bit string length to external FUs and receive 32-bit fitness values. This trait causes that the proposed processor be able to optimize problems with very large search spaces.

Comparing to the related works and the software, the proposed GAP benefits from very low calculation time. This is due to the five different techniques we have used to serve as speed boosters for the proposed GAP, which are as follows.

1) *Novel Multi-GAP Approach:* As mentioned before, several of the proposed GAPs can serially connect to each other. This feature causes the simultaneous work of processors while requires a little time for transmitting data between processors. Despite of that, the software needs to perform all the calculations in serial, which leads to a more time consuming process.
2) *Master–Slave Processing:* The proposed GAP can work with two FUs in parallel where the processor is master and the two FUs are slaves. In this mode, the fitness computation time is nearly halved.
3) *Coarse-Grained Parallel Processing:* The proposed processor supports distributed parallel processing of genetic populations. In this fashion, by employing m processors, m parallel populations can grow concurrently. This condition can be assumed as the state where there is an m-times larger population.
4) *Pipelined Fitness Computation:* As mentioned before, fitness calculations happen in external FUs. The proposed GAP can work with pipelined FUs, and supports up to 15 stages latency.
5) Built-in speed-up methods as described briefly in the following.
    a) *Pipeline Implementation of the Proposed GAP:* All modules in the GAP run at the same time and do not wait for their turn in the GA cycle.
    b) *Novel Dual-Population Scheme:* Instead of using one population causing serial execution of selection and replacement, two populations are employed where each of them runs selection or replacement in contrary of the other at the same time.
    c) *Parallel Selection:* In the selection phase, two chromosomes are selected as the winner to produce the next generation. In the proposed processor, the two winner chromosomes are obtained in parallel.

The mechanisms of implementing these techniques are explained in the next sections. The built-in methods are thoroughly explained in [22].

The proposed GAP supports one-point, two-point, and uniform crossovers, beside its support of one-point and uniform mutations. The number of inputs in the proposed GAP is such that the parameters of GA can be tuned with relatively good precision compared to other hardware GAs. In addition, by setting the proper inputs, some optional modules of the proposed GAP might be turned off or turned on during the GA procedure. From this point of view, the proposed GAP is flexible. Moreover, the GAP benefits from the random immigrants operator, which extends exploration of GA. Random immigrants operator enters random chromosomes into the population to maintain its diversity. One of the main limitations in applying GA to real systems, especially to control systems, is constraints in search space and solutions. Discarding improper solutions is a common way in optimizing constrained problems. Although discarding is supported in the proposed GAP, user instead may define penalty terms to the fitness of infeasible solutions in external FUs, making them less likely or impossible to impact on the population.

The rest of this paper is organized as follows. Section II gives a brief introduction about the implemented GA. Section III describes behavioral design of the proposed digital GAP and explains its functions. Section IV reports simulation results. Section V concludes with a brief summary of the results and characteristics of the proposed GAP.

II. Implemented Genetic Algorithm

In the following, basic concepts about the realized GA model are discussed.

A. Steady-State Genetic Algorithm

Steady-state GA [23] replaces new individuals in the population as soon as their fitness values are evaluated. Since there is no interrupt in the steady-state GAs procedure, it is suitable for pipelined processing. Another prominent type of GA is generational GA [23]. Generational GA collects new offspring to produce a new population (generation) and when a fresh generation is generated replaces it on the previous population. The concept of generation stalls the pipeline procedure, because in generation replacement phase other operators such as selection, crossover, mutation, and fitness computation would not operate [9]. In [24], it is claimed that the absence of generation in steady-state GA makes it more appropriate for parallel processing. It is found in [25] that steady-state mechanism improves GAs tracking ability in online applications.

Considering the aforementioned advantages, the steady-state GA has been implemented in the proposed GAP. Pseudocode of the steady-state GA is represented in the following.

```
GENERATE anInitialPopulation
CALCULATE theFitnessOfInitialPopulation
IF stoppingConditionIsNotMet
    PROCESS selectionAndChooseParents
    PROCESS crossoverOnTheSelectedParents
    PROCESS mutationOnTheOffspring
    CALCULATE theFitnessOfOffspring
    REPLACE newOffspringInThePopulation
ENDIF
RETURN theBestFoundChromosomeAsTheOutput
```

B. Delete-the-Oldest Replacement

There are two popular replacement strategies in steady-state GA, namely delete-the-oldest and delete-the-worst. Delete-the-oldest replaces new chromosomes on the oldest individuals in the current population while delete-the-worst overwrites offspring on the chromosome with lowest fitness value. Delete-the-worst replacement is more likely to find better results, but it totally cannot track the system changes [25], which are crucial in online circumstances. Therefore, delete-the-oldest is chosen as replacement strategy of the proposed GAP. Delete-the-oldest requires a counter to hold the address of the oldest chromosome in the population. Every time replacement occurs, it counts and the next chromosome would be prepared to be overwritten in next stage. Thus, delete-the-oldest strategy is simple and fast unlike delete-the-worst that requires a min (or max) circuit to find the lowest fitness in the population. Min (or max) circuits must compare all the individuals in the population to find the one with lowest (or highest) fitness (or cost). In contrast to the counting action in a counter, this procedure is time-consuming and dependent on the number of individuals in the population.

III. Behavioral Design

A. Pipeline Scheme

Hardware GAPs have the opportunity to benefit from pipeline structure, which makes them faster. In the proposed method, GA operations proceed in pipeline fashion as demonstrated in Fig. 1. Middle of the block diagram shows that selection, crossover and mutation (as a part), fitness computation, and replacement modules are pipelined. In every repeat, they operate concurrently and their outputs will be fed to the successor in the next iteration. Hence, duration of every loop of GA is equal to the time that is required by the slowest module. This is more time-efficient compared to the nonpipelined GA where every loop has a processing time equal to the sum of the time each module requires. Main loop modules tune their concurrency asynchronously [i.e., no clock (CK) is used to synchronize modules]. Each module has a completion flag and when all modules raise their completion flag, a new iteration starts and resets the modules. The proposed GAP starts up by the initialization module. Initialization resets the processor, generates random chromosomes, and simultaneously

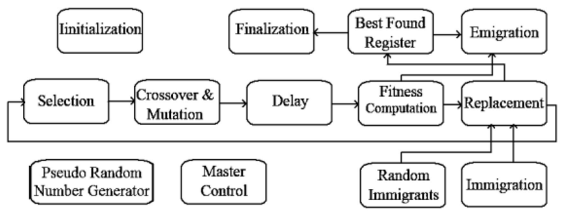


Fig. 1. Block diagram of the proposed GAP.

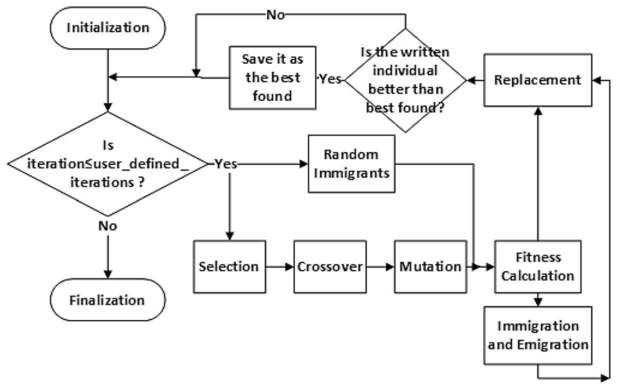

Initialization
No
Save it as the best found
Yes
Is the written individual better than best found?
Replacement
Is iteration≤user_defined_iterations ?
Yes
Random Immigrants
No
Selection
Crossover
Mutation
Fitness Calculation
Finalization
Immigration and Emigration

Fig. 2. Flowchart of the genetic functions in the design.

uses the fitness computation and replacement modules to create a population of evaluated chromosomes. After the whole population was created, master control module takes the control from initialization module and the pipelined loop of the GA starts gradually. Pipeline loop modules start iteration by iteration from selection to replacement, because for example there is no offspring generated in the first iteration for the replacement module to write them in the memory. Immigration and emigration modules are for coarse-grained parallel processing. Immigration takes chromosomes from other GAPs and emigration sends selected chromosomes out. These two modules work in pipeline with the modules they interact. In addition, the random immigrants module, which is responsible for inserting random chromosomes into the population, works in pipeline with replacement module. Furthermore, in Fig. 1, the pseudorandom number generator (PRNG) provides random bits everywhere the randomness is needed. The best-found register saves the best chromosome to be sent out as the result at the end of GA procedure. Every time a chromosome is written to the memory (replaced in the population), it would be compared with the best-found chromosome to be overwritten on the best found if it has larger fitness value. Eventually, when the GA reaches its stopping condition, which is a fixed number of iterations set by user, finalization module takes the control of the processor and puts the best-found chromosome on the 32-bit data bus as the output. Finalization module also resets the rest of the processor for power saving considerations. The flowchart of the genetic functions in the pipeline loop is illustrated in Fig. 2.

B. General Purpose Processor

As mentioned before, the proposed GAP is independent from specific applications. For this purpose, the fitness

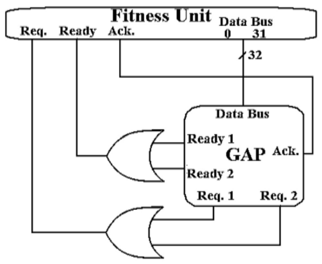


Fig. 3. Handshake connection of the GAP and the external FU.

TABLE I
COMPARISON BETWEEN DUAL-POPULATION SCHEME AND STANDARD STEADY-STATE SINGLE POPULATION GA

	Fitness of dual-population scheme	Fitness of standard genetic algorithm
Function 1	Fitness=$\lvert(1-x)\times x^2\times\sin(200\pi x)\rvert$	
Best fitness in 100 runs	0.1481	0.1481
Average fitness in 100 runs	0.1479	0.1477
Function 2	Fitness=$(1-(2\times\sin^{20}(3\pi x))+\sin^{20}(20\pi x))^{20}$	
Best fitness in 100 runs	1.0486×10^6	1.0486×10^6
Average fitness in 100 runs	1.0481×10^6	1.0473×10^6

computation module itself does not contain any evaluation circuits. Rather, it evaluates the chromosomes through external FUs connected to the GAP by handshake connection, as shown in Fig. 3. When fitness computation module starts, it initiates a source-initiated handshaking and places the first offspring on the 32-bit data bus, then raises the request 1 (Req. 1). After receiving acknowledge (Ack.) signal (Ack. goes to high), it lowers Req. 1 and repeats the same procedure for the second offspring. A similar approach is used then for receiving the fitness values of chromosomes sent to the FU, based on destination-initiated handshaking. Ready pin goes to high and the GAP waits for calculated fitness and its corresponding Ack. signal. After receiving the second fitness value, fitness computation module finishes it operations. This procedure is repeated in each iteration of GA.

C. Dual-Population Scheme

There is a critical time in the pipelined steady-state GA procedure, when both selection and replacement modules try to access the memory for reading and writing, respectively. Although the proposed GAP can perform read and write operations serially (if user set the proper control input), a new method called dual-population scheme is designed to parallelize them. By using two populations, in each iteration, parents would be chosen from individuals of one population and the new chromosomes replace on the other population. In the next iteration, populations swap their role, for example the one used for selection works in replacement module. In this way, the pipeline does not stall and selection and replacement modules work in parallel. However, one issue is that, in each iteration, the new children only go to one of the populations. Hence, half the generated chromosomes will exist in one population and another half in the other. In order to prevent isolation of the two populations, a delay module was added in the GAs normal loop as shown in Fig. 1. Addition of the delay module causes that the number of modules in the normal loop of GA be odd. Knowing this and knowing that the populations swap their function in selection and replacement in every iteration, it is worth mentioning that the produced chromosomes from one population replace in the other one. Therefore, the populations have mutual interactions. Note that, since there is no selection in the initialization, in that phase both populations are used in replacement module, causing identical chromosomes replace on them.

It is obvious that the GA using dual-population scheme is different from the original GA. In order to compare their robustness in finding better results, the two abovementioned GA methods were implemented in MATLAB software while their parameter specifications was the same. Table I shows the obtained results. It is clear that the dual-population scheme achieved better results compared to the original GA that is due to the existence of twin populations (with twice memory requirement) in the dual-population scheme. Anyhow, more investigation is needed to completely verify the efficacy of dual-population scheme, which is beyond the scope of this paper.

The main advantage of the dual-population scheme is its compatibility with pipeline structure. In our design, replacement module requires 4 CK cycles for its operations and selection needs 3, 5, 9, or 17 CK cycles to finish, based on the tournament size inputs set by user. Because the typical GA performs serial operation of selection and replacement it would require 7, 9, 13, or 21 CK cycles to finish these modules. On the other hand, selection and replacement in the dual-population scheme are executed in parallel and the maximum of 4, 5, 9, or 17 CKs are needed to finish both modules. Therefore, average processing time of replacement and selection modules in the dual-population scheme is $((4/7) + (5/9) + (9/13) + (17/21))/(4) = 0.657 \cong 65\%$ of the typical GA case. To conclude this section, dual-population scheme needs more examination to substantiate its functionality. Therefore, the option of standard steady-state GA is also available for use in the GAP.

D. Parallel Tournament Selection

Tournament selection [23] method has indispensable impact on the overall GA. The roulette wheel and the tournament are the most well-known methods used for selection of parent chromosomes. Roulette wheel selection [9] has been used in many GA implementations, but it has an intrinsic loop that stalls the pipeline structure of GA [9] and makes it not suitable for a fast GAP. On the other hand, for selecting each parent, tournament selection chooses c random chromosomes from population and selects the one with higher fitness. Hence, it only requires a max operation and is far faster than the roulette wheel selection. Therefore, it is used in the proposed GAP. The number of contestants (c) adjusts selection pressure. Higher contestant numbers prolongs the selection procedure.

One important feature of the tournament selection is that the selection of two parents can be carried out in parallel by using two max circuits and two memories to be read from. By utilizing a twice memory mechanism, our design benefits from

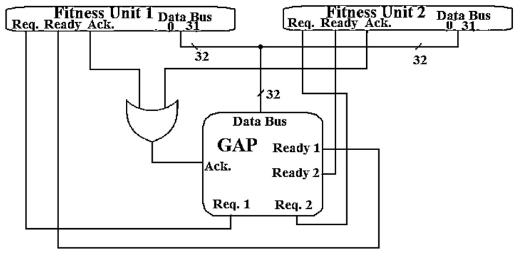


Fig. 4. Handshake connection of the GAP and two external FUs in master–slave parallel processing.

this parallel parent selection. That is, in the replacement phase, new individuals are written identically in two memories and then in the selection phase each of the two memories are read separately and provide contestants for selecting the two parents in the max circuits. Therefore, the parallel processing reduces the selection time by half. Nevertheless, this approach necessitates twice memory hardware like the dual-population scheme. Regarding the aforementioned techniques, the proposed GAP possesses four random access memories (RAMs). The fourfold memory structure is cost of the speed-up techniques in replacement and selection modules.

E. Master–Slave Parallel Processing

The proposed GAP is capable to work with two external FUs instead of one FU to evaluate fitness (or cost). Basically, there is no need for further FUs, because in each repetition of the algorithm only two chromosomes are evaluated. Master–slave processing with two FUs decreases the time spent for fitness calculation, which has major impact on the total processing time of the GAP. If $t_{\text{fitness\_calculation}} \gg t_{\text{communication}}$ master–slave processing nearly doubles the speed of fitness computation process, where $t_{\text{fitness\_calculation}}$ is fitness computation duration of each FU and $t_{\text{communication}}$ is communication interval between the GAP and FUs. Fig. 4 shows the connection of FUs to the GAP in master–slave mode. Compared to Fig. 3, the communication procedure does not change except the existence of two FUs.

F. Coarse-Grained Parallel Processing

In coarse-grained processing, several populations grow in parallel on GAPs. They transmit chromosomes to each other in a predetermined frequency of iterations. Therefore, any processor must have emigration and immigration modules for sending and receiving migrant chromosomes. In order to maintain the number of communication lines as few as possible, serial data transfer is adopted. By using quad I/O half-duplex serial peripheral interface (SPI), immigration and emigration modules have four data lines, a CK, and a chip select (CS) line. The proposed processor has one emigration and two immigration modules. Thereby, in a repetition, it can receive immigrant chromosomes from two other GAPs while sending its best-found chromosome or first offspring to any other GAPs in the network. Fig. 5 demonstrates a sample connection of GAPs. Network topology is not restricted to the chain model and various topologies are practicable. Note that the frequency

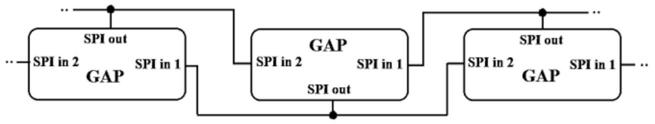


Fig. 5. Chain connection in coarse-grained parallel processing.

of chromosome transfer (iteration gap between two transmissions) is adjustable by user in our design. Also, by setting the proper input, second immigration module can be turned off.

If activated by user, emigration and immigration modules start to work every m iterations after the initialization period, where m is the frequency of migration. Concerning the synchronization of transfers and prevent blind sending, every time emigration starts it waits until all input CSs go to high and then begins to send its migrant chromosome. The reason is that CS output rises only when the processor is ready to both send and receive migrant chromosomes. If the second immigration module is turned off by user, emigration module only monitors the first immigration module's CS input. Immigration module raises its completion flag when its input CS goes to high and then lowered. When a CS connection is lowered, it means that chromosome transfer in that iteration is finished. Received chromosomes from other GAPs are passed to replacement module to be written in the next iteration.

G. Elitism

Elitism is an efficient way for hastening the convergence of the algorithm to the optimum by preventing the elimination of best chromosomes. In our design, the best-found individual would not be deleted. This is accomplished by participating the chromosome stored in best-found register in each tournament selection whether the best found exist in the population or not. The probability of selecting the best-found register's chromosome is $(1/p)$, where p is the population number. Thus, probability of selecting individuals stored in the populations (RAMs) is $((p-1)/p) \times (1/p) = ((p-1)/p^2)$. Note that elitism can be turned off by user.

H. Random Immigrants

One of the ways for increasing exploration of the algorithm and make it more effective in dynamic environments is to use random immigrants operator. It introduces new genetic material in every time step [26], thus making it an option to avoid fast convergence of the algorithm. A survey of the recent uses of this approach can be found in [26]. If user sets the proper input and activates this operator, GAP executes it with a frequency of iterations, which is set by user (via dedicated inputs), too. Every time random immigrants module starts in a repetition of the algorithm, a random chromosome is generated and replaces second offspring before it goes to fitness computation module. The new random individual then continues the normal loop of GA by going to fitness computation and replacement modules.

I. Improper Solutions Discard

When working to solve constraint problems, discard operator may be a good choice. If an external FU supports this operation, it is possible to raise the discard input of the GAP if the FU detects an infeasible solution. After sending the chromosome, the GAP raises its ready output and then FU replies to this destination-initiated handshaking by raising discard and acknowledge inputs of GAP. By receiving a discard signal, fitness computation module zero outs the ready flag of corresponding chromosome. Whenever an individual's ready flag is zero, it will not be written on memories in the replacement module and actually will be eliminated.

J. *Stop and Restart*

Restart input is used for startup or resetting the GA operations. The GAP would not begin GA until it receives restart signal, which resets all parts and starts the GA. When working, a restart signal from outside the processor implies the reset of all parts and restarting the GA. This may be useful when working with very unsteady systems, because perhaps it is better to restart the GA instead of waiting for GA itself track the system changes (i.e., changes in the fitness function). In fact, restarting the algorithm may be considered as the simplest way of addressing dynamic optimization problems [27]. The stop input is devised as well for applications that their sampling time is less than the time required by the GAP. Whenever this input becomes high, GA will halt, almost all parts reset, and the best-found solution will be placed on the 32-bit data bus.

K. *Pipeline Scheme for Fitness Computation*

Fitness calculation is among the most time consuming operations in GA. Proceeding it with the pipeline fashion significantly decreases delay but introduces latency. The proposed GAP is able to work with FUs with up to 15 stages of latency. Assuming that the communication time between FUs and the GAP is pretty less than the fitness computation time, we will have *s* times speed-up in pipeline processing of fitness calculation, where *s* is the latency of pipeline. Latency is defined to the GAP by setting a specific 4-bit input by user. When the GAP operates in the pipelined fitness evaluation, it keeps any chromosome sent to external FU by entering the chromosome to a first-in-first-out (FIFO) memory. Each time a new chromosome enters the FIFO memory a shift of data toward the output of FIFO happens. The GAP synchronizes the chromosome pull out from FIFO with its corresponding calculated fitness, obtained from external FU, by means of the user specified latency.

L. *Multi-GAP: Extending the Search Space*

Multi-GAP is a way to use several connected GAPs in problems with more than 32 bits. A bit extending method is presented in [16]. Its main disadvantage is in crossover where always a crossover occurs between two GA cores. Also, in its mutation, each core performs a one-point mutation by itself, which is different with *n*-point mutation. In [28], another method for extending the search bits is presented. It introduces an interesting technique, but has many differences with the standard GA. Our design is such that there is no difference between the single and multi-GAP formations. Multi-GAP is

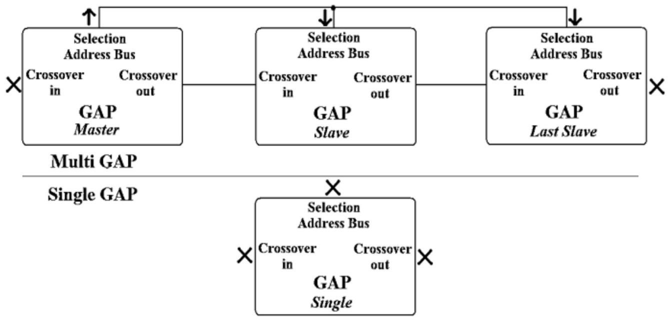


Fig. 6. Classification of GAPs according to their position in the multi-GAP chain.

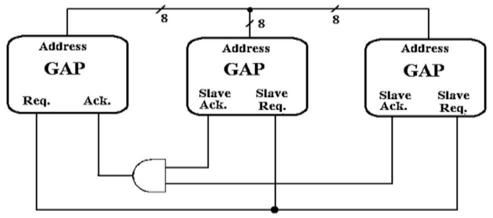


Fig. 7. Connection of address buses in a three-GAP example.

formed by connecting the processors in a chain, as demonstrated in Fig. 6. First processor is master and the last one must be set as last slave. Intermediate processors are named slave. A 2-bit input is used to define the GAPs as master, slave, last slave, or simply in single mode. Single mode is the default mode in which all multi-GAP features are deactivated. In multi-GAP, each processor produces its part of the chromosome string, going to external FU, while all receive the same fitness value from FU. Therefore, selection operator cannot be done on all the processors, because all slices of chromosome string must be obtained from one selection operation. Thus, master processor determines the parents and sends their memory address to the slave and last slave processors through the address bus. Address bus, shown in Fig. 6, is a sender while the GAP is set to master, is a receiver in slave modes, and is turned off in single chip mode. It transfers address data with handshake protocol. Fig. 7 shows address bus connection in a three-GAP sample. AND gate causes request signal of the master remain high until all slaves rise their acknowledge outputs.

In contrast to mutation and uniform crossover, one- and two-point crossover operations require communications between GAPs, since if a crossover happens in a point, all bits from that point to the end of parent strings must be swapped. Therefore, crossover sender and receiver handshake ports are devised, so that a GAP notifies swap of parents to the next processor if it experiences a one-point crossover. As illustrated in Fig. 6, master just sends the crossover state since it has no precedent as well as its crossover state receiver is always off. Slave processors use both ports, whereas in the last slave just crossover receiver is active.

In order to decide the location of crossover in one- or two-point modes, a 2-bit input is added to the proposed GAP. If both or just 1-bit of crossover type input is high,

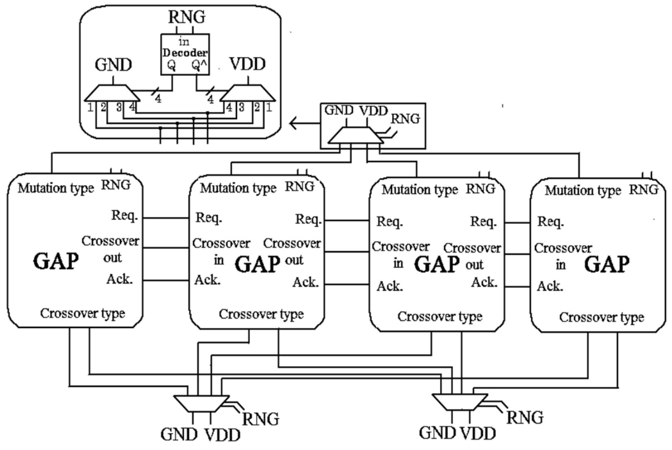


Fig. 8. Random appointment of two-point crossover and one-point mutation to four GAPs.

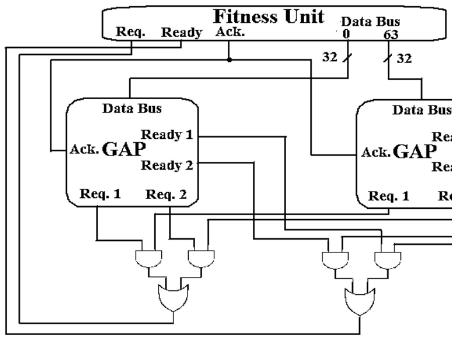


Fig. 9. Connection type and required external fitness circuitry for a two-GAP implementation.

then a two-point or one-point crossover is performed in that GAP, respectively. Likewise, no one- or two-point crossover is executed in the absence of high value in crossover type inputs. The bottom of Fig. 8 shows the implementation of two-point crossover in a four-GAP example. Using demultiplexers, only two of eight crossover type input bits are raised and the rest are grounded. Hence, with a probability, two one-point or a two-point crossover is expected. The probability comes from RNG outputs of GAPs, which determine demultiplexer states. In every repetition of GA, the processors refresh their RNG outputs with new random bits. Note that, in the aforementioned example, one-point crossover was achievable by utilizing a demultiplexer only.

Like uniform crossover, uniform mutation does not need to external circuits. In the case of one-point mutation, the top of Fig. 8 illustrates a formation similar to the one-point crossover. Mutation type bit specifies that a one-point mutation is performed (if be high) or not (if be grounded). Note that, structures shown in Fig. 8 are just for multi-GAP and in single mode, fix voltage values are enough for crossover and mutation type inputs. Also, note that the aforementioned crossover and mutation type inputs differ from the inputs responsible of activating uniform crossover and mutation.

Connection with external FUs in multi-GAP also needs attention. A two-GAP formation sample associated with a FU is illustrated in Fig. 9. Each GAP places its 32-bit chromosome

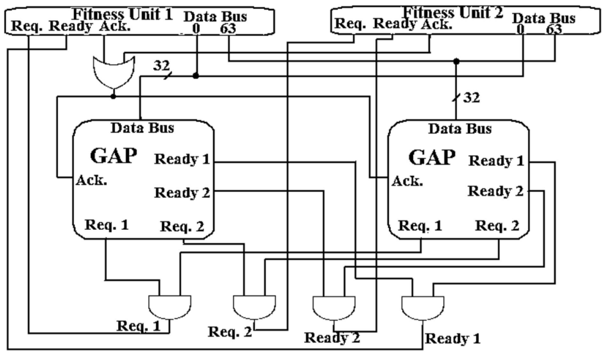


Fig. 10. Connection type and required external fitness circuitry for a two-GAP master–slave implementation.

share on the 64-bit data bus. On the other side, FU sends back identical fitness values on each 32-bit data bus. As shown in Fig. 9, the procedure is same as the single-GAP (shown in Fig. 3), except that equivalent outputs of GAPs must simultaneously be high to raise the proper input of FU (due to the existence of AND gates). Fig. 10 demonstrates sample connections of the two-GAP example which is applied to the master–slave (double FU) processing. The architecture can be clarified regarding the aforementioned descriptions (Fig. 9) and explanations about master–slave processing (Fig. 4). Note that in the case of the presence of discard in the architecture it would have the same wiring and gating as acknowledge have.

M. Flexibility in Parameter Setting

In the proposed GAP, population size can be set to 32, 64, 128, and 256, while user may choose the maximum number of 16, 32, 64, 128, 256, 512, 1024, and 2048 iterations for GA. Chromosome bit length is adjustable to 16 bits instead of 32 to improve the speed and power consumption profile in applications with search spaces equal or fewer than 16 bits. The same is true for fitness bit length. In these two cases, speed increases when emigration module transfers fewer bits, and power is saved due to the working of only 16 bits in the processor. Tournament size options are 2, 4, 8, and 16. In the proposed GAP, there is a uniform crossover input that when it is high the processor conducts uniform crossover, otherwise the 2-bit crossover type pins determine the one-point, two-point, or no crossover to be applied. A similar uniform mutation input is predicted which in the case of ground state relegates the mutation to mutation type pin to specify the one-point or no mutation, as described in the previous section. Additionally, for crossover and mutation there is two separate 7-bit inputs to specify the crossover/mutation rate from (1/128) to (128/128). A 2-bit input sets the frequency of running random immigrants module with 1, 2, and 4 iterations or no run at all. The mutation rate and random immigrants frequency are the tools in the proposed GAP to control the exploration of the GA in contrast to its exploitation property. A 4-bit input is responsible for setting the pipeline latency of fitness computation from no pipeline to 15 stages. Frequency of iterations for sending immigrant chromosomes to other GAPs is also tunable by a 2-bit input, which can be one,

TABLE II
PROCESSING TIME OF THE MAIN MODULES IN DIFFERENT CONFIGURATIONS

Module	**Processing time in different configurations**			
Selection (including memory access for reading)	**2 contestants**	**4 contestants**	**8 contestants**	**16 contestants**
	$3 \times CK_1$*	$5 \times CK_1$	$9 \times CK_1$	$17 \times CK_1$
Crossover and Mutation	**Single-GAP**	**Multi-GAP**		
		Uniform Crossover	**One-point / Two-point Crossover**	
	$4 \times CK_1$	$4 \times CK_1$	$\mathrm{Max}\{4 \times CK_1, CK_1 + (n^{**} \times Communications)\}$	
Fitness Calculation and Random Immigrants	**One fitness unit**		**Master-Slave processing (two fitness units)**	
	$\frac{Fitness\_calculation\_time}{s^{***}} + Communications$		$\frac{Fitness\_calculation\_time}{2 \times s} + Communications$	
Replacement (including memory access for writing)	$4 \times CK_1$			
Emigration and Immigration	**16-bit chromosome**		**32-bit chromosome**	
	16-bit fitness	**32-bit fitness**	**16-bit fitness**	**32-bit fitness**
	$8 \times CK_2$****	$12 \times CK_2$	$12 \times CK_2$	$16 \times CK_2$

* First clock's delay (CK_1) is equal to 2.2 ns.
** n is the number of GAPs in multi-GAP structure minus one.
*** s is the pipeline stages plus one.
**** Second clock's delay (CK_2) is equal to 1.3 ns.

two, and four GA iterations or simply no migration. Also, there is a pin to choose between first offspring and best found chromosome to be sent out in emigration module. The proposed processor contains 16 seeds for its PRNG that each of them may be selected by a 4-bit input. Starting with different seeds is essential in multi-GAP and coarse-grained processing where multiple processors work concurrently, because identical behavior is expected with equal seeding.

N. Processing Time Analysis

The main motivation of this paper is to hasten the execution time of GA. In the following, Table II expresses the processing time of the main modules in different configurations.

Communication times are the delays required for establishing handshake connections. In the proposed pipeline structure, the overall processing time in each iteration of the algorithm will be equal to the delay of the slowest module. In one exception, when dual-population scheme is disabled, replacement and selection modules work serially.

IV. RESULTS AND DISCUSSION

The proposed GAP was implemented in CSMC 0.18 μm process and simulated with HSIM and CustomSim software packages. In order to measure speed, power consumption, and convergence performance of the GAP, five test benches were devised. At first, we look at the two test functions used in the first four experiments.

A. Test Functions

Two types of external FUs, which calculate fitness values based on the Knapsack Problem and Royal Road Function rules were designed ideally in Verilog-A language. External FUs' response time were estimated from their longest data path in a real circuit implementation. The aforementioned test functions are described in the following.

1) Bounded Knapsack Problem: Knapsack problem is one of the combinatorial optimization problems. Some of its variants are considered among NP-Complete (nondeterministic polynomial time-complete) problems. Given a set of items with predefined weight and value, knapsack problem is to find a subset with as much as value that does not exceeds a specific weight limit. We have used a sample of its bounded version

$$\text{Fitness} = \text{Max}\left(\sum_{i=9}^{11} v_i \times x_i + \sum_{i=17}^{19} v_i \times x_i + \sum_{i=25}^{27} v_i \times x_i + \sum_{i=41}^{43} v_i \times x_i + \sum_{i=57}^{59} v_i \times x_i\right) \leq 450 \quad (1)$$

where $v_i = i$ and $x_i \in \{0,\ 1,\ 2,\ 3\}$. So far, it is a kind of bounded knapsack problem, but another limitation was added to ease the hardware structure. That is, for every summation operator ($\sum_i^{i+2} v \times x$), there is a

$$x_i + x_{i+1} + x_{i+2} \leq 3 \quad (2)$$

condition. Hence, in each sigma operator, up to three summations can be done. This means that, as an example for the first sigma operator in (1), from the set of numbers 9, 10, and 11 at most three repeatable numbers should be selected. Some examples are {9}, {9, 10}, and {10, 10, 11}.

Because v_i values are predefined in (1), GAP just optimizes the x_i. From (1), every sigma operator has three dedicated x_i s, each one corresponds to a v_i. In addition, from (2) each sigma operator at most has three inner summations. Therefore, for every sigma operator three 2-bit connections between the GAP and FU would be sufficient. Each 2-bit is for an inner summation. For example in the first sigma operator, 00 for each 2-bit represents no summation, while 01, 10, and 11 represent 9, 10, and 11, respectively. This mechanism drives to a 30-bit data bus for transferring chromosomes (because of the five 6-bit sigma operators). To return the fitness value, FU uses 9 pins out of 30, because the maximum fitness is less than 450. If the fitness value exceeds 450 then FU raises its discard output. Additionally, Ack., Req., and ready pins are placed in the FU for communication with the GAP.

After defining the problem, Fig. 11 shows a plot of fitness values of individuals in the steps of 10^6. Although depicting only a small part of the search space, this illustration exhibits the large amount of local maxima and minima in the fitness, making it very hard for the algorithm to find the global optimum.

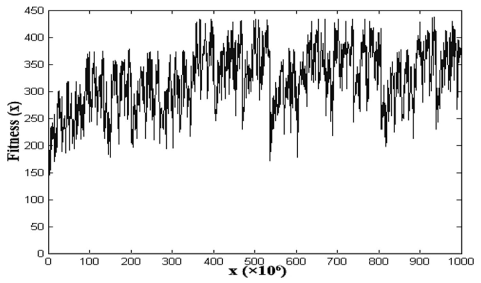


Fig. 11. Fitness function of the bounded knapsack problem.

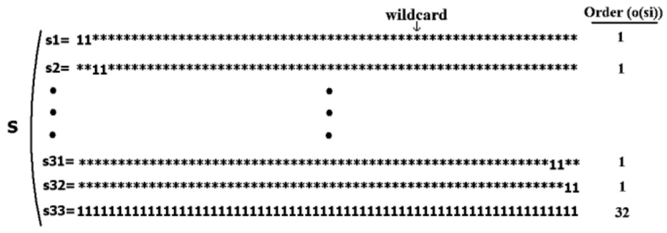

wildcard
Order (o(si))
S
s1= 11**
1
s2= **11**
1
s31= **11**
1
s32= **11
1
s33=1111111111111111111111111111111111111111111111111111111111111111
32

Fig. 12. Royal road landscapes in the tests.

2) Royal Road Function: Royal road function, introduced in [29], is one of the most important ways to evaluate GAs. In the function's principles, GA should seek the optimum solution by finding fitness landscapes and then deploying them. Due to its discrete local optimums and high steepness in fitness landscapes, royal road is very hard to be solved. The modified function in our tests is shown as follows:

$$\text{Cost}(x) = 64 - \sum_{i=1}^{33} \delta_i(x) \times o(s_i),$$

$$\text{where } \delta_i(x) = \begin{cases} 1 & \text{if } x \in s_i \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where s_i stands for a bit string with some specified bits while the rest of the bits in the string are don't-care. An input string x would be a sample of a s_i (i.e., $x \in s_i$) when it contains the same specified bits in s_i. Multiplying factor o(s_i) also determines the importance of each s_i. Fig. 12 illustrates the predefined landscapes and their o(s_i) factor. Star sign demonstrates a wildcard in the string. All shown landscapes in Fig. 12 have 64 bits, thus it is mandatory for chromosomes to have 64 bits, too. To provide 64-bit chromosomes, multi-GAP structure (actually with two GAPs) is required. Nevertheless, the external FU itself should possess 64 pins to receive chromosomes and in return place cost values on seven of them (cost value is between 0 and 64). It also has Ack., Ready, and Req. pins to communicate with the GAP.

Fig. 13 demonstrates Cost*(x)* values in 1000 points of *x*. It has a great number of local optima. From the figure, it can be seen that there is no cost lower than 50. This, again, stresses the difficulty of lessening the cost in this problem.

B. Simulation Results

Five test sets with various configurations for the GAP are to be described in this section. Configurations of the tests are just for verifying the implemented hardware, and not for achieving the best possible results. In general, the first one is to validate the GAP functioning. The second and third tests prove the coarse-grained and multi-GAP processing, respectively. Almost all the functions of the GAP are tested in the fourth test. Finally, the fifth experiment tests the proposed GAP in a practical application. Respecting the tests, detailed configurations of the GAP inputs are shown in Table III.

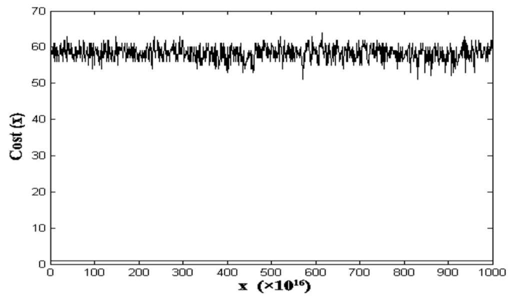


Fig. 13. Cost function of the royal road function.

1) First Test: In the first test, a GAP tries to solve the previously explained knapsack problem with the assistance of two external FUs. Chromosome length is 32, because at least 30 bits are needed to transfer chromosomes between GAP and FUs. Fitness length is likewise 16, since the number of required bits for fitness transmission is 9.

Fig. 14 depicts simulation results. The top signal is for the best-found fitness value over time, and the middle one stands for data ready output. Data ready went to high after 2.903 μs since the start of the GAP, signaling the end of operations. Output was a chromosome with a fitness equal to 450. Hence, the algorithm found the optimum. The bottom of Fig. 14 is replacement module's start signal. Whenever it is high, replacement module is working. It is shown to give a better understanding about pipeline arrangement in the GAs main loop. From that, it can be deduced that replacement module runs repeatedly in initialization phase. After initialization, master control module holds it off until all the preceding modules in the pipeline loop are started. Following the non-activity period, replacement module recurrently operates in the normal loop of GA until the GAP finishes operations. It should be noted that the total power consumption of the GAP was 3.36 W.

In order to appraise the results, an identical test set, written in C language, provided a comparative measure. C code was run on a 2.5 GHz dual-core machine. Average fitness and elapsed time per 100 runs of algorithm were 449.65 and 715.39 μs, respectively. Therefore, the proposed GAP is not only 246 times faster but it also achieved a more robust solution.

One of the main properties of every GA is its repeatability. Fig. 15 investigates the fitness improvement of 16 runs of the GA in this test. Each run was characterized by its PRNG seed. Results show that in all 16 runs, fitness improvement virtually followed a similar pattern, which may be a necessity in some applications.

2) Second Test: Second test aims to verify the coarse-grained parallel processing of GAPs in solving the bounded knapsack problem. In the test, there are two GAPs exchanging their best-found chromosome every two iterations, by just

TABLE III
CONFIGURATIONS OF THE TEST BENCHES

Test	Master-slave processing	Chromosome bit length	Fitness bit length	Pipelined fitness computation	Multi-GAP processing
1	Yes	32	16	No	No
2	Yes	32	16	No	No
3	No	64	16	Yes, two stages latency	Yes, two processors
4	Yes	256	16	Yes, five stages latency	Yes, eight processors
Test	**Dual-population scheme**	**Elitism**	**Random immigrants**	**Tournament selection size**	**Population**
1	Yes	Yes	Yes, every 4 iterations	4	128
2	No	No	No	2	64
3	Yes	Yes	Yes, every iteration	8	32
4	Yes	Yes	Yes, every 2 iterations	16	256
Test	**Iteration number**	**Crossover type**	**Crossover rate**	**Mutation type**	**Mutation rate**
1	128	Two point	100/128	One point	7/128
2	16	Uniform	113/128	Uniform	3/128
3	32	One point	111/128	One point	25/128
4	256	Uniform	89/128	Uniform	11/128
Test	**Coarse-grained parallel processing**	**Chromosome to be sent in parallel processing**	**Frequency of parallel processing**	**Fitness or cost**	**Test function**
1	No	-	-	Fitness	Knapsack Problem
2	Yes, two GAPs	Best found	Every 2 iterations	Fitness	Knapsack Problem
3	No	-	-	Cost	Royal Road (64-bit)
4	Yes, four sets of multi-GAPs	First offspring	Every 4 iterations	Cost	Royal Road (256-bit)

* Configurations of the fifth test is similar to the first test, except the chromosome bit length that is 16 and the test function, which is an image enhancement application.

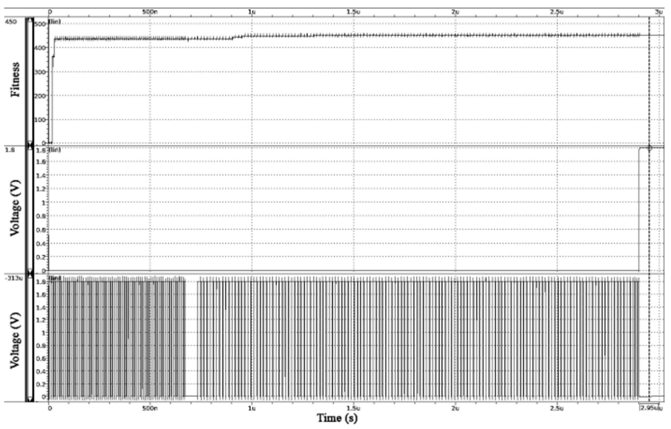


Fig. 14. Results of the first test.

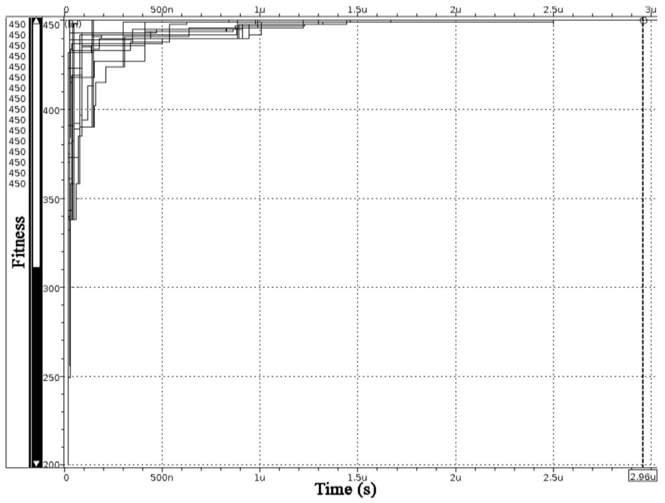


Fig. 15. Repeatability test of the algorithm in 16 runs of the first test.

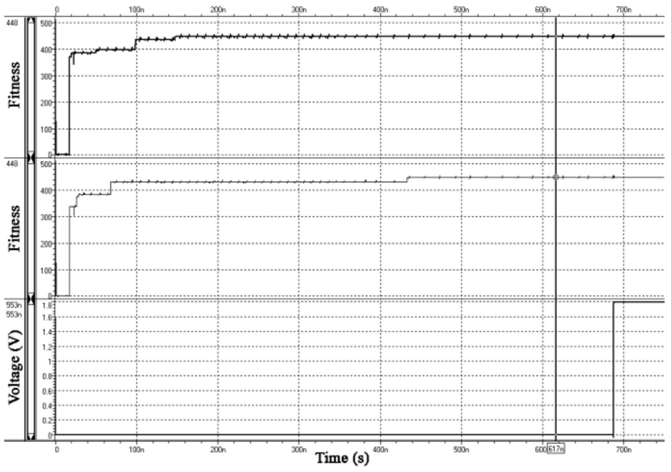


Fig. 16. Results of the second test.

one of their SPI ports. Each GAP itself possesses two subordinate external FUs, and initiates with a different PRNG seed. Dual-population scheme is also disabled, making the algorithm slower and identical to the original steady-state GA.

The top two signals in Fig. 16 depict best chromosomes finding progress of the two GAPs participated in the test. Final result is 448. It is recognizable that around 433 ns, the best fitness of the second GAP improves suddenly. The reason is that it received an immigrant fitter than its best found, thus replaced it with the new one. Beneath the best found signals in Fig. 16 are overlapping data ready signals of the GAPs. It shows that both GAPs ended their operations on 687 ns.

In the second test, a C coded software counterpart was developed and executed on the same computer mentioned earlier. The average fitness and duration for 100 runs were 446.41 and 613.31 μs, respectively. Another time, the GAP achieved better result and gave 892.73$\times$ speed-up in the process.

3) Third Test: In this test, royal road is the function to be optimized by two processors in a multi-GAP fashion. Both processors interact with a 64-bit external FU by their 32-bit data buses. Although, either send their portion of 64-bit chromosome to FU, they get the same cost value. In this way,

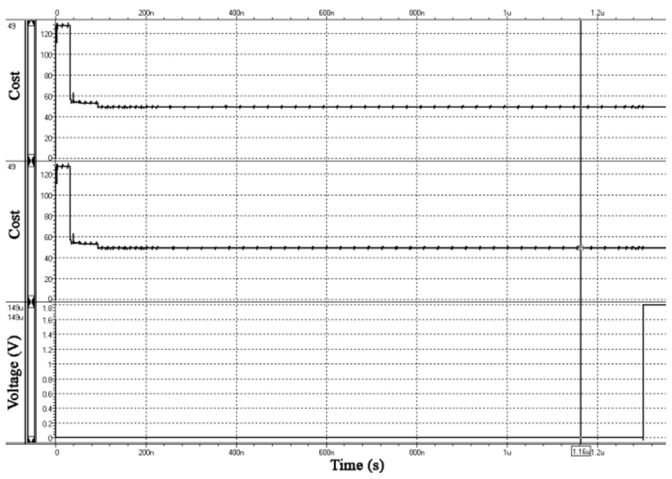


Fig. 17. Results of the third test.

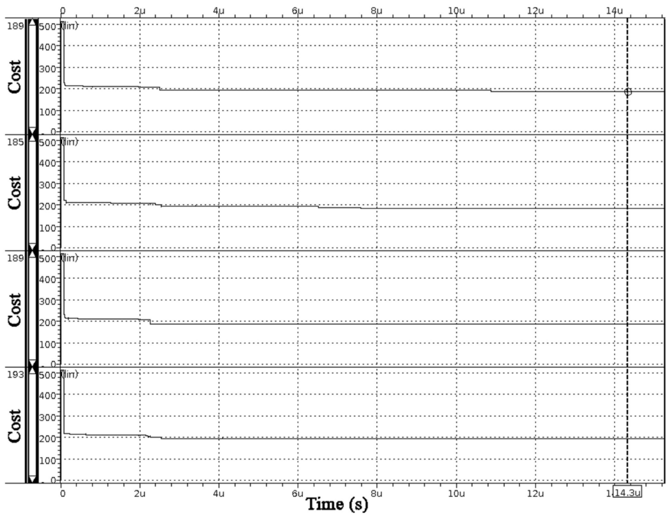


Fig. 18. Cost reduction progress in the fourth test.

Fig. 9 illustrates the GAP chain linked to external FU. In this test, FU has two pipeline stages. The first processor in multi-GAP chain plays the master role, while the next is last slave. Connections required for selection is akin to those shown in Fig. 7, though without the third GAP and AND gate. Connections for crossover and mutation are modified versions of those in Fig. 8. Instead of four GAPs, two 4-output demultiplexers for crossover, and one another for mutation, here we have just two GAPs and a dedicated two-output demultiplexer for either crossover or mutation (due to one-point crossover, one-point mutation, and two existing processors). It should be noted that two different seeds were utilized in the processors to give them distinctive performance.

Fig. 17 in its top two signals demonstrates how best cost for the two GAPs decreased throughout the operations. Since, both processors receive an identical cost value, their procedure is also similar. The final best cost was 49, and as the bottom signal (data ready signal) shows operations concluded at 1301 ns. The probability of gaining a cost of 49 or lesser, in the test, was nearly 9×10^{-9}. We can see that the GAP was able to reach this small portion of good solutions.

Once again, a software, with conditions like previous tests, is employed to bring comparative results. On average, software reduced the cost to 47.59 in 362.15 μs. Although, hardware obtained a weaker solution; however, it proceeded operations 278.36 times faster. It is obvious that in the third test, speed-up was less than previous tests. It can be explained that, generally multi-GAP structure with one/two-point crossover delays the procedure, because crossover and mutation data must propagate among GAPs. In multi-GAP, crossover and mutation will have no negative impact on processing time if they are adjusted for uniform modes.

4) Fourth Test: A comparatively comprehensive benchmark with 32 GAPs was designed in the fourth test. In this test, four sub-populations evolved simultaneously. Topology of the coarse-grained parallel processing was a neighborhood (bidirectional ring) structure. Aim of the test was to optimize a 256-bit version of royal road function. Thereby, eight processors were connected in each subpopulation. To accelerate calculations, dual-population scheme, master–slave processing, and pipelined fitness computation were enabled. Cost reduction plots of the four subpopulations are depicted in Fig. 18. When all the processors ended their operations at 14.431 μs, average best-found cost of the four subpopulations was 189. Knowing that the probability of having Cost $\leq$ 189 is approximately 8×10^{-86} in the current test, we observe that the 8×10^{-86} of the best results are found in less than 15 μs.

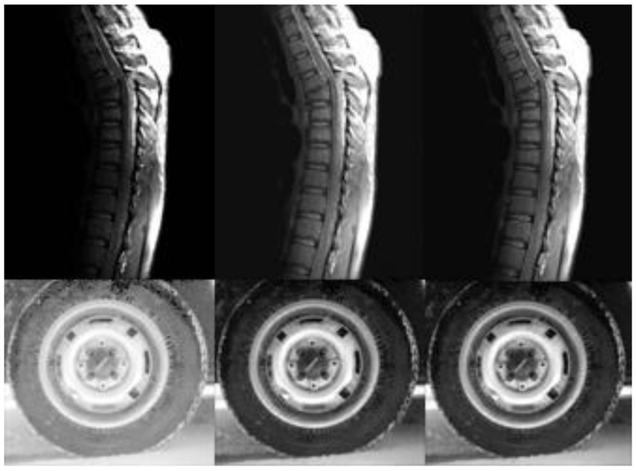
The software counterpart, mentioned in the previous tests, must perform all the calculations alone, making it very hard to compete with 32 GAPs that work concurrently. The software does not benefit from two parallel populations or parallel fitness computation. It is restricted by its nature to not use pipelined processing. Consequently, it finished its operations in 77.8 ms with an average fitness of 191.99 in its subpopulations. Comparing with the results obtained from the proposed GAP, speed-up over software is 5391, which obviously is in favor of the proposed GAP.

5) Fifth Test (Real-World Application): References [30] and [31] use GAs in real-world applications. In order to examine the efficacy of the proposed hardware in solving real problems, we prepared an image processing test application. The objective of the application is to enhance contrast of images by using a simple power-law (gamma) transformation. Gamma transformation [32] performs an exponentiation operation on every pixel intensity value in the image. The transformation in our test is

$$s = c \times r^{\frac{\gamma}{k}} \tag{4}$$

where r is pixel intensity, c and k are constants equal to 1 and 10 000, and γ is the exponent to be determined, by GA, for a relatively better contrast than the original image. The FU in our benchmark comprises the previously explained gamma transformation and appraisal of its fitness. The fitness function used is thoroughly explained in [33]. It uses standard deviation and entropy of intensity values in image as well as an edge detection operator (Sobel [32]), as shown in the following:

$$\text{Fitness} = \rho \times \sqrt[3]{\text{STD} \times \text{Entropy} \times \text{Sobel}};\ \ \rho = 1000. \tag{5}$$



(a) (b) (c)

Fig. 19. Results of performing the fifth test. (a) Original image. (b) Output image of software. (c) Output image of the proposed GAP.

TABLE IV
FITNESS AND CHROMOSOMES OBTAINED IN THE FIFTH TEST

Image	Software (MATLAB code)		Hardware (the proposed GAP)	
	Chromosome	Fitness	Chromosome	Fitness
Spine	5073	19452	5870	26200
Tire	26005	27741	26600	36400

By employing the specific FU in this test, we aimed to show the applicability of the proposed GAP in real-world problems, not to show the effectiveness of the fitness criterion itself in solving image enhancement problems. The configuration of the single GAP in this test is similar to the first test, except the chromosome and fitness bit lengths. Both of them are in the range of 0–65 536 and fixed to 16 bits. In addition, there is no discard action in the current test, because no constraints are defined.

The results obtained from performing the test on two sample images are demonstrated in Fig. 19. The output images of the proposed hardware are compared to the results of an identical software code. The only difference between the hardware and the software is in the use of dual-population scheme in the proposed GAP. The images in Fig. 19 reveal that software and the proposed hardware perform similarly. The fitness values and the associated chromosomes (γ values) are presented in Table IV. Again, it can be seen that the software and the proposed hardware approximately behave alike, even though the GAP obtains slightly better objective results.

C. Comparison With Related Works

There is no widely accepted way to directly compare GAPs. This shortcoming is mostly due to the incompatibility in functions provided and possible configurations of GAPs. For instance, an application-specific GAP restricts the comparison with only processors in that application. Another example is difference in available options for implemented GAs. In actuality, it is impractical to set the same benchmark to compare this paper with all other works. Therefore, in the absence of a universal benchmark, speed boosting over software has been the only numerical comparison tool. Table V shows speed-up results obtained in the previous works, beside the present GAP.

TABLE V
SPEED-UP TEST RESULTS COMPARISON WITH PREVIOUS WORKS

Hardware platform	Chromosome bit length	Parallel processing	Pipelined fitness computation	Software host machine [a]	Speedup over software	Relative Speedup	Work
0.5μm (sim[b])	64	No	No	N/A	10-20	-	[2]
FPGA (imp)	N/A	No	No	1.6 GHz	10.53	16.84	[5]
					12.06	19.29	
					250	400	
FPGA (imp)	N/A	N/A	No	2.4 GHz	10.68	25.63	[6]
FPGA (imp)	8	No	Yes	333 MHz	468	152.19	[9]
	30				221	71.86	
FPGA (sim)	4	No	No	50 MHz	37-102	1.81-4.98	[13]
FPGA (sim)	N/A	No	No	440 MHz	≈100	42.96	[14]
FPGA (imp)	94	No	Yes	100 MHz	2200	214.84	[15]
	70			366 MHz	320	114.37	
FPGA (imp)	16	No	No	400 MHz [c]	5.16	2.01	[16]
FPGA (imp)	N/A	No	No	12.5 MHz	110.71	1.35	[17]
					110.92	1.35	
					112.09	1.36	
					1.24	0.01	
FPGA (sim)	32-256	Yes	No	2 × 2.2 GHz	77-111	338.8-488.4	[18]
FPGA (imp)	32	No	No	2 × 2.09 GHz	48-90	200.64-376.2	[19]
0.25μm (sim)	1000	No	No	400 MHz	200	78.12	[34]
0.35μm (sim)	1000	No	No	400 MHz	130	50.78	[35]
0.18μm (sim)	32	No	No	2 × 2.5 GHz	246.43	1232.15	First test
	32	Yes	No		892.73	4463.65	Second test
	64	No	Yes		278.36	1391.8	Third test
	256	Yes	Yes		5391.17	26955.85	Fourth test

[a] Programming language for all works is C/C++, except [2], [6], [14], [17], and [34], which are unknown.
[b] sim stands for simulation, imp for implementation.
[c] PowerPC405 processor in Xilinx Virtex-II Pro

Comparing the achieved results with the previous works in Table V, we see a great improvement in speed. To better demonstrate the effective hardware acceleration, we defined a relative speed-up as follows:

$$\text{Relative Speed-up} = \text{speed-up} \times \text{processor} \tag{6}$$

where speed-up is those shown for each work in Table V, and processor is the CPU power of software's host machine in GHz. Relative speed-up measures the hardware relative acceleration by considering software's computational power. Regarding the compared works, their relative speed-ups are

TABLE VI
COMPARISON OF THE AVAILABLE FUNCTIONS AND ADJUSTMENTS

Work	**Chromosome bit length**	**Selection method**	**Mutation method**	**General purpose**	**Fitness bit length**
Prpsd.	Unlimited (multi-GAP), Each GAP: 16, 32	Tournament, with 2, 4, 8, or 16 contestants	Uniform, 1-point	Yes	16, 32
[2]	64	Roulette wheel	N/A	Yes	16
[5]	8 - 1024	Tournament, with 2 contestants	N/A	Yes	8 - 1024
[16]	Unlimited (multiple GA cores), Each GA core: 16	Roulette wheel	1-point	Yes	16
[17]	N/A	Roulette wheel	Uniform	Yes	N/A
Work	**Simultaneous selection and replacement**	**Population size**	**Crossover method**	**Coarse-grained parallel processing**	**Parallel parent selection**
Prpsd.	Yes, by using dual-population scheme	32, 64, 128, 256	Uniform, 1-point, 2-point	Yes, adjustable	Yes
[2]	N/A (Simple GA [a])	64, 128	Adaptive, Uniform, 2-point	No	No
[5]	N/A	8 - 16384	Uniform, 1-point, 2-point, Cross-point	No	No
[16]	N/A (Simple GA [a])	1 - 256	1-point	No	No
[17]	N/A (Simple GA [a])	N/A	2-point	No	Yes
Work	**Iteration number**	**Operators for control or online applications**	**Pipelined fitness computation**	**No. of possible fitness units**	**No. of Crossover rates**
Prpsd.	16, 32, 64, 128, 256, 512, 1024, 2048	Random immigrants, Bad solutions discard	Yes, up to 15 pipeline stages	2	129
[2]	512, 1024, 2048, 4096	-	No	2	64
[5]	N/A	-	No	N/A	N/A
[16]	1 - 2^{32}	-	No	1	16
[17]	N/A	-	No	1	N/A
Work	**RNG seeds**	**Parameters adjustable after fabrication**	**Elitism**	**Pipelined genetic algorithm**	**No. of Mutation rates**
Prpsd.	16 predefined true random seeds	Yes	Yes	Yes	129
[2]	-	Yes	Yes	Yes	256
[5]	-	No (ASIC), Reprogram needed (FPGA)	No	No	16384
[16]	Programmable, 3 built-in seeds	Yes	Yes	N/A	16
[17]	N/A	Reprogram needed (FPGA)	No	No	N/A

[a] Inherently, in generational (simple) GA, selection and replacement do not operate synchronously.

also shown in Table V. Table V exhibits speed superiority of the proposed work. For instance, the fastest previously reported work's [18] relative speed-up is equal to 488.4, which is 9.14 times slower than the proposed GAP (it was compared with the second test results due to mutual similarity in the test features). However, dissimilarities in hardware platform and technology nodes are also important. It is obvious that some works in Table V are implemented in older technology nodes, such as 0.5, 0.35, and 0.25 μm. This makes the comparison harder, because a fast technology platform is a plus for doing rapid calculations. All in all, comparing ASIC implementations in Table V, speed superiority of the proposed work is much more than the expected improvement of just technology enhancement.

In addition to speed, adjustability is another advantage of the proposed GAP, especially if we take inherent hardware limitations into consideration when we speak of making changes in GA parameters. The proposed GAP is general-purpose and has many different adjustment inputs to provide diversity in the available GA (see Section III-M). Comparing the recent works, which we have enough information about their flexibility in parameters, Table VI proves the advantage of the proposed design from the point of flexibility in parameter adjustment and available functions, even in contrast to FPGA-based counterparts [5], [16], [17]. The method explained in [5] is also very flexible, but before synthesizing its Verilog code. After that, there is no way to modify GA parameters. Furthermore, the provided software to adjust GA parameters before implementation in [5] lacks options for defining mutation type and crossover rate. Concerning the maximum chromosome bit length, Table VI shows that there is a bit length extension method proposed in [16]. According to that method, a group of GA cores uses a shared memory to access chromosomes. However, it has difficulties with crossover, where at all connection points between two GA cores an unwanted crossover occurs incorrectly. In addition, it independently performs one-point mutation in every GA core, which differs from *n*-point mutation.

D. Discussion

GA requires many parameters to be set prior to its start up. The proposed processor is flexible in setting the parameters of the algorithm. In this manner, user can fine tune the parameters of GA and turn optional modules on or off. It effectively supplies a variety of prevalent GA arrangements. In the proposed design, GA parameters are changed by setting proper inputs, not by redesign or reprogram.

In addition to the speed, power consumption and maximum functionality of the realized GA were among important considerations in the design. Power consumption of the presented processor is about 3.3 W. It comprised 478 079 MOS transistors and has 129 input–output pins. Apart from the utilized speed-up techniques, the proposed GAP benefits from transistor-level design. In contrast to high-level design platforms, like hardware description languages that provide quick development process, direct transistor-level implementations are better suited for enhancements such as speed boosting, etc. To the best of the authors' knowledge, the proposed work is the only GAP in the literature directly realized on transistor-level circuit. All the previous works were based on the synthesizing of hardware description languages.

V. Conclusion

In this paper, a hardware steady-state GAP in 0.18 μm ASIC process was presented. Its replacement model is delete-the-oldest. Intrinsically, it has elitism, random immigrants, and bad solutions discard operators. In addition, penalty for bad solutions is supported if it is included in external FU. Additionally, a group of the proposed processors can work in parallel, while occasionally exchanging chromosomes (coarse-grained parallel processing). The provided parallel processing is an ideal tool to hasten genetic evolution. The proposed design is not restricted to solution space magnitude. In the design, every processor can have 16- or 32-bit lengths and several of them might be chained to render a larger multiple of 16-bit length. By performing GA operations in pipeline, beside parallel parent selection and the novel dual-population scheme, the proposed GAP is essentially rapid. Simulation results showed that the presented design is far faster than the earlier works. Notwithstanding the improvements in quickness and quality of results, user can turn dual-population scheme off to stick to the original steady-state GA. The proposed GAP works fine with two fitness computation units and is able to benefit from pipelined fitness computation (up to 15 stages). Its use is not bounded to specific applications.

One of the future works could be to examine the dual-population scheme thoroughly, and its effect on the GA results. Extending the hardware GA to have some existing evolutionary features like the migration models expressed in [36] and [37] for artificial bee colony and the coevolution is also interesting for future works.

References

[1] S. D. Chen, P. Y. Chen, and Y. M. Wang, "A flexible genetic algorithm chip," in *Proc. National Comp. Symp.*, Taipei, Taiwan, 2003, pp. 253–257.

[2] S. Wakabayashi *et al.*, "GAA: A VLSI genetic algorithm accelerator with on-the-fly adaptation of crossover operators," in *Proc. IEEE Int. Symp. Circuits Syst.*, Monterey, CA, USA, 1998, pp. 268–271.

[3] N. Yoshida and T. Yasuoka, "Multi-GAP: Parallel and distributed genetic algorithms in VLSI," in *Proc. Syst., Man, Cybern.*, Tokyo, Japan, 1999, pp. 571–576.

[4] K. Kobayashi, N. Yoshida, and S. Narazaki, "GAP/D: VLSI hardware for parallel and adaptive distributed genetic algorithms," in *Proc. Int. Joint Conf. Comput. Sci. Opt.*, Sanya, China, Apr. 2009, pp. 95–98.

[5] P. Y. Chen, R. D. Chen, Y. P. Chang, L. S. Shieh, and H. A. Malki, "Hardware implementation for a genetic algorithm," *IEEE Trans. Instrum. Meas.*, vol. 57, no. 4, pp. 699–705, Apr. 2008.

[6] W. Tang and L. Yip, "Hardware implementation of genetic algorithms using FPGA," in *Proc. 47th Midwest Symp. Circuits Syst.*, Hiroshima, Japan, Jul. 2004, pp. 549–552.

[7] S. D. Scott, A. Samal, and S. Seth, "HGA: A hardware-based genetic algorithm," in *Proc. 3rd Int. ACM Symp. Field-Program. Gate Arrays*, New York, NY, USA, 1995, pp. 53–59.

[8] Y. H. Choi and D. J. Chung, "VLSI processor of parallel genetic algorithm," in *Proc. 2nd IEEE Asia Pac. Conf. ASICs*, 2000, pp. 143–146.

[9] O. Kitaura *et al.*, "A custom computing machine for genetic algorithms without pipeline stalls," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, Tokyo, Japan, 1999, pp. 577–584.

[10] M. S. Ben Ameur, A. Sakly, and A. Mtibaa, "Implementation of real coded genetic algorithms using FPGA technology," in *Proc. 10th Int. Multi-Conf. Syst. Signals Devices*, Hammamet, Tunisia, 2013, pp. 1–6.

[11] R. Faraji and H. R. Naji, "An efficient crossover architecture for hardware parallel implementation of genetic algorithm," *Neurocomputing*, vol. 128, pp. 316–327, Mar. 2014.

[12] J. Kok, L. F. Gonzalez, and N. Kelson, "FPGA implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 272–281, Apr. 2013.

[13] V. P. Nambiar, S. Balakrishnan, M. Khalil-Hani, and M. N. Marsono, "HW/SW co-design of reconfigurable hardware-based genetic algorithm in FPGAs applicable to a variety of problems," *Computing*, vol. 95, no. 9, pp. 863–896, Sep. 2013.

[14] G. Koonar, S. Areibi, and M. Moussa, "Hardware implementation of genetic algorithms for VLSI CAD design," in *Proc. Int. Conf. Comp. App. Ind. Eng.*, San Diego, CA, USA, 2002, pp. 197–200.

[15] B. Shackleford *et al.*, "A high-performance, pipelined, FPGA-based genetic algorithm machine," *Genet. Program. Evol. Mach.*, vol. 2, no. 1, pp. 33–60, Mar. 2001.

[16] P. R. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica, "Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine," *IEEE Trans. Evol. Comput.*, vol. 14, no. 1, pp. 133–149, Feb. 2010.

[17] N. Nedjah and L. D. M. Mourelle, "An efficient problem independent hardware implementation of genetic algorithms," *Neurocomputing*, vol. 71, nos. 1–3, pp. 88–94, Dec. 2007.

[18] Y. Jewajinda, "A performance evaluation of a probabilistic parallel genetic algorithm: FPGA vs. multi-core processor," in *Proc. Int. Comp. Sci. Eng. Conf.*, Nakorn Pathom, Thailand, 2013, pp. 298–301.

[19] M. A. Moreno-Armendariz, N. Cruz-Cortes, C. A. Duchanoy, A. Leon-Javier, and R. Quintero, "Hardware implementation of the elitist compact genetic algorithm using cellular automata pseudo-random number generator," *Comput. Elect. Eng.*, vol. 39, no. 4, pp. 1367–1379, May 2013.

[20] D. V. Coury, R. P. M. Silva, A. C. B. Delbem, and M. V. G. Casseb, "Programmable logic design of a compact genetic algorithm for phasor estimation in real-time," *Elect. Power Syst. Res.*, vol. 107, pp. 109–118, Feb. 2014.

[21] E. Mininno, F. Neri, F. Cupertino, and D. Naso, "Compact differential evolution," *IEEE Trans. Evol. Comput.*, vol. 15, no. 1, pp. 32–54, Feb. 2011.

[22] P. Hoseini, A. Khoei, K. Hadidi, and S. Moshfe, "Fast and flexible genetic algorithm processor," in *Proc. 18th IEEE Int. Conf. Electron., Circuits, Syst.*, Beirut, Lebanon, 2011, pp. 635–638.

[23] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998.

[24] M. Nowostawski and R. Poli, "Parallel genetic algorithm taxonomy," in *Proc. 3rd Int. Conf. Knowl.-Based Intell. Inf. Eng. Syst.*, Adelaide, SA, Australia, 1999, pp. 88–92.

[25] F. Vavak and T. C. Fogarty, "Comparison of steady state and generational genetic algorithms for use in nonstationary environments," in *Proc. IEEE Int. Conf. Evol. Comput.*, Nagoya, Japan, May 1996, pp. 192–195.

[26] C. Cruz, J. R. Gonzalez, and D. A. Pelta, "Optimization in dynamic environments: A survey on problems, methods and measures," *Soft Comput.*, vol. 15, no. 7, pp. 1427–1448, Dec. 2010.

[27] S. Yang, H. Cheng, and F. Wang, "Genetic algorithms with immigrants and memory schemes for dynamic shortest path routing problems in mobile ad hoc networks," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 40, no. 1, pp. 52–63, Jan. 2010.

[28] M. Salami, *Multiple Genetic Algorithm Processor for Hardware Optimization* (LNCS 1259). Berlin, Germany: Springer, 1997, pp. 247–259.

[29] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and GA performance," in *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. Cambridge, MA, USA: MIT Press, 1992.

[30] W. Fu, M. Johnston, and M. Zhang, "Low-level feature extraction for edge detection using genetic programming," *IEEE Trans. Cybern.*, vol. 44, no. 8, pp. 1459–1472, Aug. 2014.

[31] Y. Yoon and Y. H. Kim, "An efficient genetic algorithm for maximum coverage deployment in wireless sensor networks," *IEEE Trans. Cybern.*, vol. 43, no. 5, pp. 1473–1483, Oct. 2013.

[32] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Harlow, U.K.: Prentice Hall, 2008.

[33] P. Hoseini and M. G. Shayesteh, "Efficient contrast enhancement of images using hybrid ant colony optimisation, genetic algorithm, and simulated annealing," *Digital Signal Process.*, vol. 23, no. 3, pp. 879–893, May 2013.

[34] T. Imai, M. Yoshikawa, H. Terai, and H. Yamauchi, "Scalable GA processor architecture and its implementation of processor-element," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, Orlando, FL, USA, May 2002, pp. 3148–3151.

[35] T. Imai, M. Yoshikawa, H. Terai, and H. Yamauchi, "VLSI processor architecture for real-time GA processing and PE-VLSI design," in *Proc. Int. Symp. Circuits Syst.*, Vancouver, BC, Canada, May 2004, pp. 625–628.

[36] S. Biswas *et al.*, "Migrating forager population in a multi-population artificial bee colony algorithm with modified perturbation schemes," in *Proc. IEEE Symp. Swarm Intell.*, Singapore, Apr. 2013, pp. 248–255.

[37] S. Biswas, S. Das, S. Debchoudhury, and S. Kundu, "Co-evolving bee colonies by forager migration: A multi-swarm based artificial bee colony algorithm for global search space," *Appl. Math. Comput.*, vol. 232, pp. 216–234, Apr. 2014.

Seyed Pourya Hoseini Alinodehi received the B.S. degree from Lahijan Branch, Azad University, Lahijan, Iran, in 2007, and the M.S. degree from Urmia University, Urmia, Iran, in 2011, both in electrical engineering and electronics.

His current research interests include computer vision, image processing, artificial intelligence, evolutionary algorithms, fuzzy systems, neural networks, and circuit design.

Sajjad Moshfe was born in Shiraz, Iran. He received the B.S. and M.S. degrees in electronic engineering from Shahid Beheshti University, Tehran, Iran, and Urmia University, Urmia, Iran, respectively, and the Ph.D. degree in electronics from Science and Research Branch, Islamic Azad University, Tehran.

He is currently with the Department of Electrical Engineering, Islamic Azad University, Arsanjan Branch, Arsanjan, Iran. His current research interests include fuzzy control and neural network systems design, optimization, and analog and digital integrated circuit design for fuzzy and neural network applications.

Masoumeh Saber Zaeimian was born in Rasht, Iran. She received the B.S. degree in electrical engineering-electronics from Islamic Azad University, Lahijan Branch, Lahijan, Iran, in 2007.

Her current research interests include power electronics and digital circuit design.

Abdollah Khoei was born in Urmia, Iran. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from North Dakota State University, Fargo, ND, USA, in 1982, 1985, and 1989, respectively.

He is currently with the Department of Electrical Engineering and Microelectronics Research Laboratory, Urmia University, Urmia. His current research interests include analog and digital integrated circuit design for fuzzy and neural network applications, fuzzy-based industrial electronics, and dc-dc converters for portable applications.

Khairollah Hadidi received the B.S. degree from the Sharif University of Technology, Tehran, Iran, the M.S. degree from Polytechnic University, New York, NY, USA, and the Ph.D. degree from the University of California, Los Angeles, Los Angeles, CA, USA, all in electrical engineering.

He is currently with Department of Electrical Engineering and Microelectronics Research Laboratory, Urmia University, Urmia, Iran. His current research interests include high-speed high-resolution data converter design, wideband integrated filter design, and nonlinearity analysis and improvement in analog circuits.