

# A Modified Steady State Genetic Algorithm Suitable for Fast Pipelined Hardware

S. Pourya Hoseini Alinodehi  
Computer Science and  
Engineering Department  
University of Nevada  
Reno, USA  
hoseini@nevada.unr.edu

Sushil J. Louis  
Computer Science and  
Engineering Department  
University of Nevada  
Reno, USA  
sushil@cse.unr.edu

Sajjad Moshfe  
Department of Electrical  
Engineering  
Islamic Azad University  
Arsanjan Branch, Iran  
s.moshfe@iaua.ac.ir

Mircea Nicolescu  
Computer Science and  
Engineering Department  
University of Nevada  
Reno, USA  
mircea@cse.unr.edu

**Abstract**— In this paper, a modification of steady state genetic algorithm, called dual-population scheme, is proposed to improve its execution speed on electronic hardware. It utilizes two memories to store two interactive populations on them. The system, inherently interchanges chromosomes between these populations. In this manner, it can fully benefit from the pipeline processing on hardware. It is shown that the proposed method performs much faster than the standard steady state and canonical genetic algorithms on pipelined genetic hardware. Moreover, the searching performance, repeatability and convergence properties of the proposed technique were tested. They show dual-population scheme performs similarly to the regular genetic algorithms while achieves better results than the present hardware-oriented genetic algorithm models.

**Keywords**— *Dual-population scheme; hardware genetic algorithm; pipeline; fast GA.*

## I. INTRODUCTION

Genetic Algorithm (GA) uses search to optimize high complexity or poorly understood problems. It is largely used when the search space is large and has a lot of local optima. Being an evolutionary computing technique, it starts with a multitude of solutions to the problem space, which based on the affinity of the approach with biology we call them chromosomes. This population of chromosomes gradually evolves as it searches for optimal solutions.

Typically, genetic algorithm's search tends to be computationally intensive, because fitness evaluations for complicated problems can be highly complex. Current interest in the internet of things (IOT) and the availability of cheap, relatively easy to use embedded processor maker kits has renewed our interest in hardware implementations of genetic algorithms that may run in an embedded environment adapting IOT things in-situ. A well-designed dedicated processor can perform GA's computations much faster than a general-purpose processor on ordinary computers. Therefore, in this paper, we investigate and evaluate a hardware implementation of genetic algorithms optimized to run on pipelined processors. The key to pipeline efficiency is the ability to run instructions in parallel.

Several hardware GA processors can be found in [1-9]. In our previous paper [1], we proposed a genetic algorithm processor that uses pipelining along with a set of other

techniques to enhance the processing speed of GA compared to software and other existing hardware implementations of GA. In its continuation, here we analyze and experimentally verify a proposed technique called dual-population scheme.

Among possible genetic algorithm implementations, the steady state genetic algorithm (SSGA) enables the most instruction level parallelism. Absence of generation in steady state GA makes it more suitable for a pipelined hardware implementation [10]. In this paper, we therefore focus on implementing a steady state genetic algorithm in hardware. Specifically, we present a modification of prior implementations of the SSGA that makes it more suitable for pipelined hardware.

There are a number of genetic algorithm variants in the literature, algorithmically designed to improve their performance on electronic hardware. The compact genetic algorithm [11] uses a probability-based representation of individuals to remove the necessity of population in GA. It, therefore, significantly reduces the memory footprint. Optimal Individual Monogenetic Algorithm (OIMGA) [12] is another proposed model of GA to diminish the memory requirements. In [13], a flavor of generational GA, called Half Sibling and a Clone (HSClone), was presented to accelerate convergence. All these hardware-oriented variations of genetic algorithms are considerably different from the well-studied canonical (generational) and steady state genetic algorithms. In contrast, we formulate a variant of steady state GA that is as close to the original well-studied one, while improving running time on pipelined hardware.

Conventionally, steady state genetic algorithm has four chief components, namely, selection, reproduction, fitness evaluation, and replacement. In a speedy pipelined GA hardware, as discussed in [1], the four sections are executed at the same time. However, the problem is that both the selection and replacement components need to access the memory containing the population of chromosomes at the same time. To overcome this issue, dual-population scheme makes it possible for parallel selection and replacement operations on the population.

In the following, section 2 introduces the proposed method and analyzes its effects from the aspect of hardware implementation and search performance. In section 3, a brief overview of the implemented hardware is given. The obtained

results are presented in section 4. Finally, section 5 concludes the paper with a discussion on what is proposed and the results.

## II. BACKGROUND AND MECHANISM

### A. Pipelining the Genetic Algorithm

In contrast to GAs run on software, hardware versions usually have separate dedicated circuits for each operation within the GA's procedure. This presents a great opportunity to increase their processing speed via simultaneous execution in a pipelined design. Fig. 1, shows a prototypical pipelined steady state genetic algorithm. It consists of selection, crossover and mutation as a single batch (reproduction), fitness evaluation, and replacement on the population. Each of the four building blocks perform their operations at the same time. As mentioned before, canonical GA have an additional generation exchange operator, which forces the pipeline structure to stop while it is being carried out [10]. Every time a fresh generation is produced, it requires selection from the new population, hence, restarts the pipeline process and imposes a new undesirable latency.

The running time of every iteration in non-pipelined SSGA is equal to the sum of the time each block takes, as follows:

$$T_i = S_i + P_i + F_i + R_i \quad (1),$$

where  $S_i$ ,  $P_i$ ,  $F_i$ , and  $R_i$  are the computation times of selection, reproduction, fitness evaluation, and replacement modules, respectively, at each iteration.  $T_i$  is the overall computation time of an iteration. Contradictorily, an ideal pipelined SSGA concludes an iteration in synchrony with the slowest module. The overall computation time during a pipelined SSGA's iteration is given by:

$$T_i = \max\{S_i, P_i, F_i, R_i\} \quad (2).$$

From (1) and (2), it is obvious that the iterations of SSGA finish sooner in a pipeline fashion. By formulating our discussion on canonical GA, we can express the time it takes for every step of a pipeline as in (3), where  $T_{population\_exchange}$  is the time required to replace the new generation on the previous one.

$$T_i \geq \max\{S_i, P_i, F_i, R_i\} + T_{population\_exchange} \quad (3)$$

By comparing (2) and (3), we can see that steady state GA may be a better fit for pipelined hardware implementation. In the following, we continue our discussion on SSGA.

### B. Enhancing the Genetic Algorithm's Pipeline

The pipelined genetic algorithm shown in Fig. 1 has a bottleneck in its procedure that can delay the time needed to finish an iteration. Both the selection and replacement components try to access the memory containing the population of chromosomes. The memory should be read from to provide data for the selection operation, while saving new evaluated

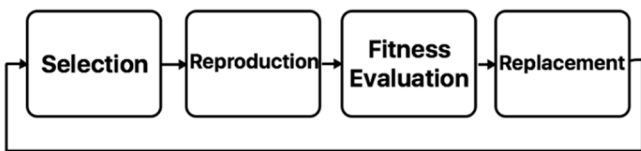


Fig. 1. A pipelined steady state genetic algorithm.

offspring in the replacement phase. Since addresses of chromosomes, being simultaneously read and written, may be identical, a dual-port random access memory (RAM) may cause data corruption. To avoid this and to generalize the concept of implementing the pipelined GA on common single-port RAMs, we propose a variation of steady state GA implementation. We expect to boost processing speed by preventing serial reading and writing of individuals, and enabling parallel reads and writes on different populations.

Dual-population scheme, as the name suggests, adds another population besides the original. In the initialization phase of the GA, initial chromosomes are written to both populations. When the main loop of GA starts, one of the two populations participates in selection, the other in replacement. In this way, the reading and writing operations in the selection and replacement components can be run concurrently. Fig. 2 demonstrates the general idea of our dual-population scheme. The two populations exchange their role in the next iteration of GA through a simple pointer swap. Consequently, each of the two populations gets new individuals every two iterations. To preclude isolation of the populations, it is sensible to have offspring produced in the first population replace members in the second population, and vice versa. This requires an odd number of pipeline stages. To accomplish this, we can break in half one of the components indicated in Fig. 1. Alternatively, we can add an extra delay module in the pipeline as shown in Fig. 3. The delay should be defined small enough to not surpass the maximum computation time of the four other components. This is necessary to maintain the running time, shown in (2), intact.

The pseudo-code of SSGA's dual-population scheme follows:

```

Initialize two identical populations, with index  $i$ ;  $\forall i = \{0,1\}$ 
 $i = 0$ 
While termination condition not met
    Select two chromosomes from population  $i$ 
    Do Crossover probabilistically
    Mutate probabilistically
    Evaluate fitness values of offspring
    Flip  $i$ 
    Replace the two offspring on population  $i$ 
End While
Output the best-found individual

```

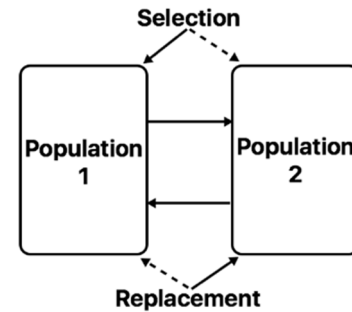


Fig. 2. Exchange of role as well as data across the selection and replacement phases in dual-population scheme.



Fig. 3. A sample pipeline dual-population GA.

### C. The Speedup Obtained Through Dual-Population Scheme

Ideal SSGA has a running time equal to that presented in (2). Nevertheless, it is not the case for practical SSGAs. Sequential reading and writing of memory, actually damages the inherent parallelism of the pipeline fashion. It turns out that the computation time of every iteration of SSGA is:

$$T_i = \max\{S_i + R_i, P_i, F_i\} \quad (4).$$

In the case of perfectly equal running times for each component of GA,  $T_i$  in (4) is two times slower than that of (2).

Distinctively, dual-population scheme is proposed to make it more suitable for the speedup achieved in a pipeline mode. The formula in (2) defines the time required to run its every iteration. To elucidate by an example, suppose the processing times of each module, taken from the practical design of our previous paper [1], is per Table I. The computation times of the tournament selection with various number of contestants are given. The fitness evaluation time depends on the problem at hand. Although it is unknown in general, we can design it in a pipeline manner with  $s$  stages to divide the time it takes by  $s$ . In [1], the maximum  $s$  is 16. Accordingly, with a master-slave style any required fitness calculation time will be divided by 32. Based on that, we assume that the fitness evaluation time is not greater than the largest processing time of the other modules, thus not affecting the overall processing time. Albeit, it is not guaranteed to satisfy this assumption for all possible fitness computing requirements. Therefore, care needs to be taken to address problems where the fitness model does not fit the current description. From Table I, we choose a configuration for the tournament selection with size of four contestants. With the above-mentioned descriptions, the running time of every iteration of dual-population GA is  $\max\{5 * CK, 4 * CK, 4 * CK\} = 5 * CK = 11 \text{ ns}$ . That would be  $\max\{(5 * CK) + (4 * CK), 4 * CK\} = 9 * CK = 19.8 \text{ ns}$  for the standard steady state GA. That amounts to  $\frac{19.8-11}{11} = 80\%$  more computation time in the case of standard SSGA compared to dual-population scheme (or 45% speedup in the case of dual-population scheme.) In the example of Table I, an average over all possible configurations yields to  $\frac{\frac{7-4}{4} + \frac{9-5}{5} + \frac{13-9}{9} + \frac{21-17}{17}}{4} = 55.7\%$  more time needed for the standard SSGA with respect to the dual-population scheme's working time, which translates to 34% speedup by using the proposed GA model.

Another important aspect to consider is that in a fixed number of iterations, the proposed method performs the same number of fitness evaluations as SSGA. In a convergence-based termination condition, though, we cannot decidedly say which of the SSGA or dual-population GA ceases tasks faster. It will be compared experimentally in the next section. Nevertheless, in the dual-population scheme the two populations are interacting by receiving offspring from each other. Thus, every

TABLE I. AN EXAMPLE ILLUSTRATING PROCESSING TIMES OF HARDWARE GA'S COMPONENTS.

Selection (Tournament)	Tournament Size			
	2	4	8	16
	$3 * CK^*$	$5 * CK$	$9 * CK$	$17 * CK$
Reproduction	$4 * CK$			
Fitness Evaluation	Single fitness unit		Master-slave (two fitness units)	
	$\frac{fitness\_calc\_time}{s^{**}} + Comm^{***}$		$\frac{fitness\_calc\_time}{2 * s} + Comm$	
Replacement	$4 * CK$			

\* Clock cycle equal to 2.2 ns.

\*\*  $s$  is the pipeline stages of fitness computation unit plus one.

\*\*\* Communication time between fitness evaluation unit and the hardware GA; It is normally low, ergo negligible, compared to the fitness calculation time.

time a population undergoes a replacement or selection, that operation will affect the other population later as well. The prevention of isolation of the populations is a key factor in avoiding halving the evolution speed. Actually, by this reasoning, it can be anticipated that the evolution speed of the proposed method be close to that of SSGA.

In addition, note that two populations double the memory footprint and, in practice, will need to be traded off against the obtained speedup.

### III. HARDWARE IMPLEMENTATION

Dual-population scheme was implemented on an Application Specific Integrated Circuit (ASIC) in the 0.18  $\mu\text{m}$  process. The general block-diagram of the realized hardware is according to Fig. 3. We chose tournament as the selection type and delete-the-oldest strategy for the replacement module. The fitness evaluation component connects to external fitness processing units through a handshake protocol. There can be up to two fitness units working as slaves with the GA processor (as master) in master-slave parallel processing mode. This way, the two new offspring in each iteration will be evaluated simultaneously. By computing fitness values in a separate part, the GA processor can be used as a general-purpose optimization hardware. The processor supports up to 15 pipeline stages in fitness calculation. In addition, the coarse-grained parallel processing feature makes it possible to transfer chromosomes to other GA processors working concurrently. Details of the hardware implementation can be found in [1].

### IV. EXPERIMENTAL RESULTS

In this section, we present the results of testing the proposed dual-population scheme along with generational GA, steady state GA, and the three hardware-oriented GAs mentioned in the first section. They are Compact GA [11], Optimal Individual Monogenetic Algorithm (OIMGA) [12], and Half Sibling and a Clone (HSClone) [13]. The tests aimed to examine the convergence property, reliability, and fitness performance of the methods. All the experiments are averaged over fifty runs. Except the last test, stopping condition of all the methods were set to probe convergence by detecting no-improvement of the best-found fitness over a predefined number of fitness

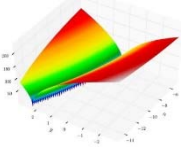
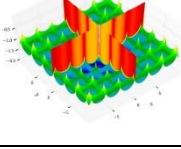
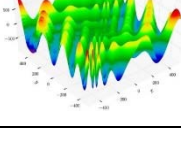
evaluations. In the following, everywhere not indicated, the crossover rate, if applicable, is 0.8, population size is 100, chromosome length is 30 bits, selection type is roulette wheel. Additionally, mutation rate for SSGA and the proposed scheme is 0.25, while it was set to 0.02 for generational GA. For OIMGA, we used  $search\_size=16$ ,  $t\_gens=6$ ,  $k\_gens=5$ ,  $d\_adjustor=4$ , and mutation rate of 0.382 [12].

We conducted our experiments on nineteen different test fitness functions and obtained similar results across these nineteen functions. Among them, we present here the results of tests on three exemplars, namely, Bukin N.6, Cross-in-tray, and Eggholder functions. They are illustrated in Table II. Bukin N.6 tests performances in a smooth search environment, while the other two possess rough and irregular search patterns.

First, we want to test how well the GAs work over different population sizes. Fig. 4, Fig. 5, and Fig. 6 show the best obtained

fitness values (negative of the costs in Table II) as an indicator of search performance, standard deviation of the found fitness values as a measure of reliability, and number of fitness evaluations before convergence, respectively. Each of the three test functions in Fig. 4 were tested in four different population sizes. By looking at the illustration we find that in eleven cases, out of twelve, the proposed method found solutions at least as good as the other methods. From the point of repeatability, dual-population scheme was among the bests in at least 10 points, out of twelve, in Fig. 5. Compact GA, OIMGA, and to some extent generational GA worked poorly in this test. Convergence test in Fig. 6 reveals that by adopting a convergence-based termination condition, dual-scheme GA performs as deficient as steady state GA in larger populations. This is considerably reduced with smaller population sizes. Again, Compact GA is the worst one in this arena.

TABLE II. TEST FUNCTIONS SHOWN IN THE FORM OF COST FUNCTIONS.

Test function	Plot	Formula	Range
Bukin N.6		$f(x, y) = 100 * \sqrt{ y - 0.01 * x^2 } + 0.01 *  x + 10 $	$-15 \leq x \leq -5$ $-3 \leq y \leq 3$
Cross-in-tray		$f(x, y) = -0.0001 * \left( \sin x * \sin y * e^{\left  100 - \frac{\sqrt{x^2 + y^2}}{\pi} \right } + 1 \right)^{0.1}$	$-10 \leq x \leq 10$ $-10 \leq y \leq 10$
Eggholder		$f(x, y) = -(y + 47) * \sin \sqrt{\left  \frac{x}{2} + (y + 47) \right } - x * \sin \sqrt{ x - (y + 47) }$	$-512 \leq x \leq 512$ $-512 \leq y \leq 512$

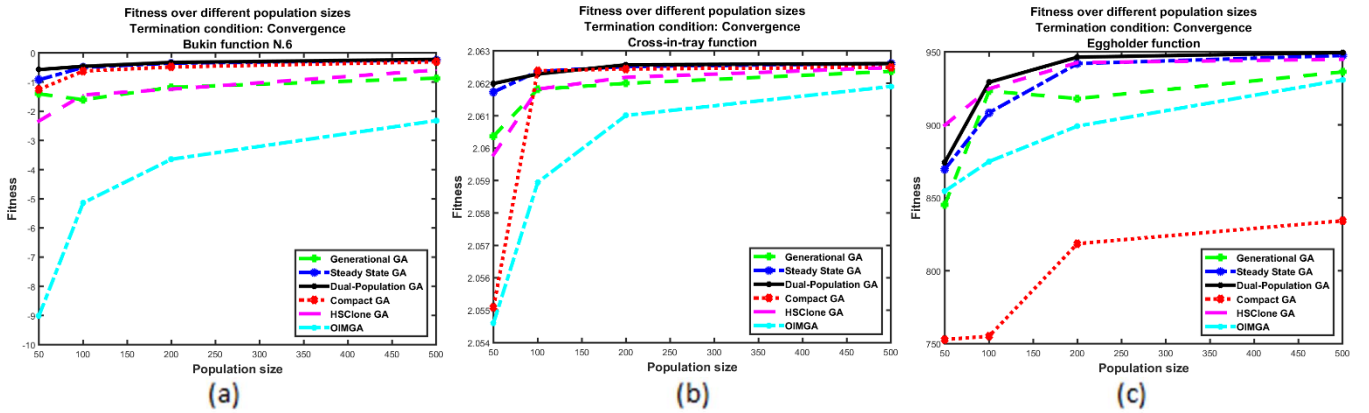


Fig. 4. Comparison of searching performances over different population sizes (higher better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

Next, the GAs are assessed over different chromosome sizes. Fig. 7, Fig. 8, and Fig. 9 demonstrate the best acquired fitness values, standard deviation of the fitness values, and fitness evaluations number before convergence, respectively. It is observed from Fig. 7 that the proposed method discovers superior fitness values in 10 points among the total 12 points.

That is the same in the case of standard deviation of the best-found optima in Fig. 8. Assuming a termination condition based on convergence, evaluation numbers over different chromosome sizes, depicted in Fig. 9, show the proposed method ceases its operations faster for the smaller chromosome sizes.

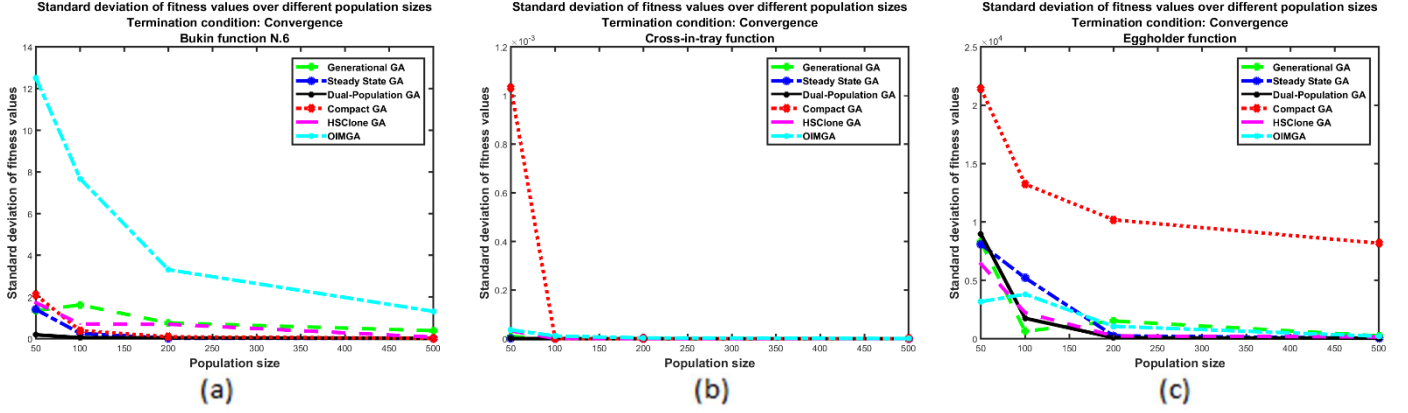


Figure 5. Comparison of repeatability in searching performance over various population sizes (lower better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

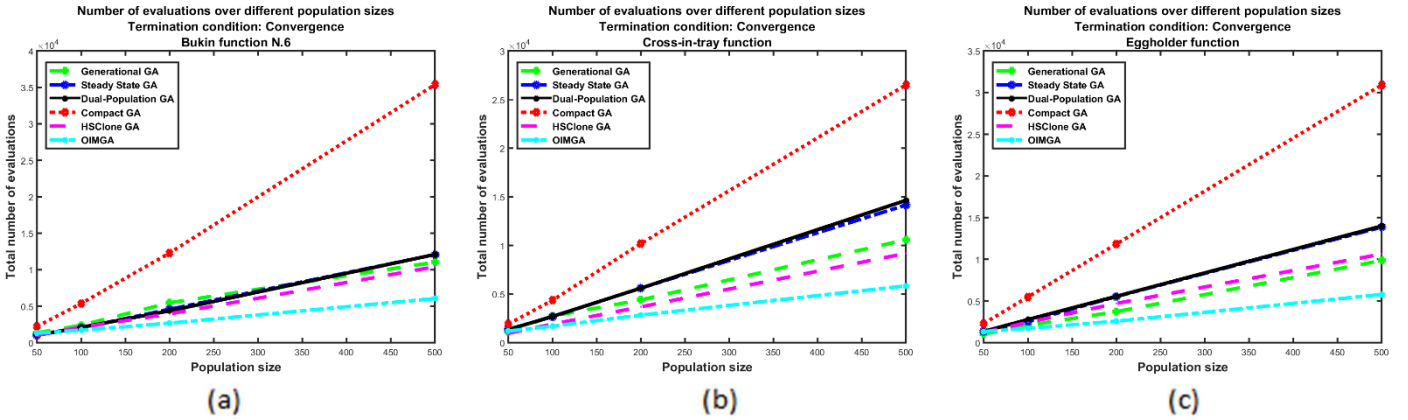


Figure 6. Comparison of convergence speed over various population sizes (lower better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

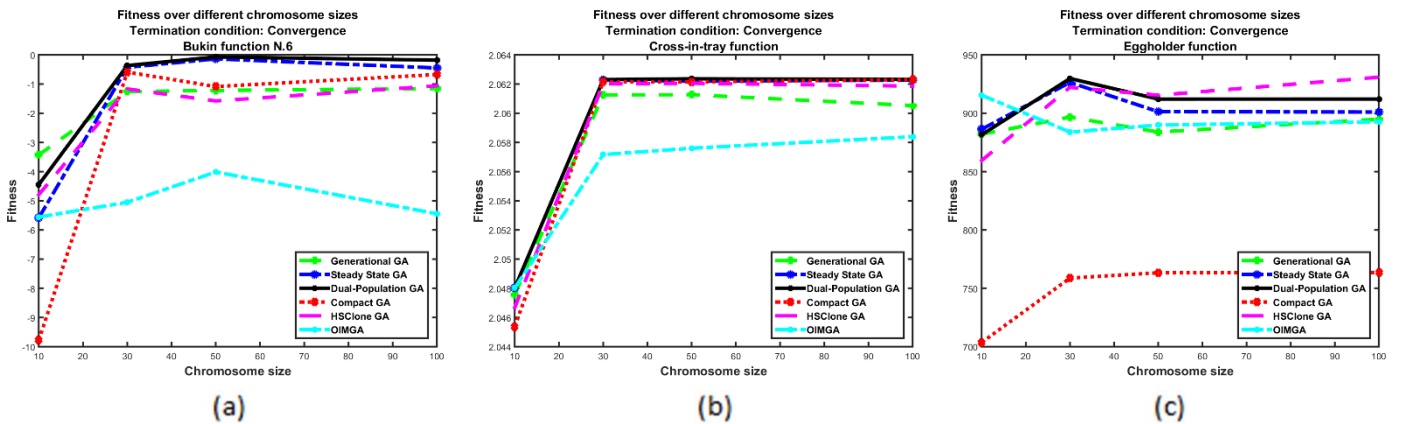


Figure 7. Comparison of searching performances over different chromosome lengths (higher better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

The search capabilities of the GAs in finding higher fitness values over different crossover rates, mutation rates, and fixed number of evaluations as the termination condition are depicted in Fig. 10, Fig. 11, and Fig. 12 in order. They approve again that dual-population GA finds optimum solutions very good in

contrast to other GAs. Among them, Fig. 12 is mostly interesting, because it discloses the efficacy of the proposed method when a fixed-iteration termination condition is being used. The obtained fitness values are almost in all cases well compared to other methods in the experiment.

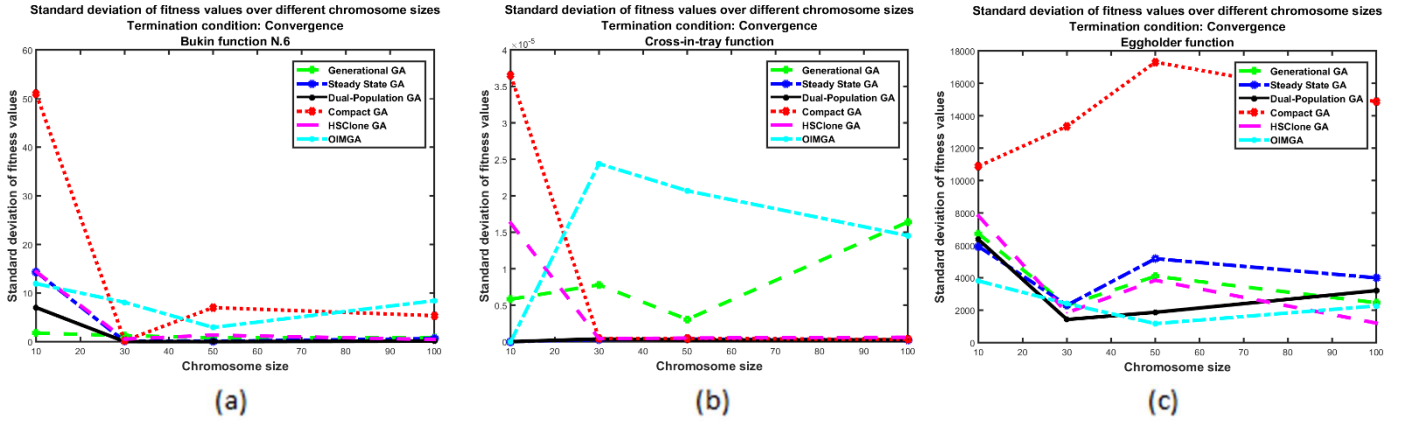


Figure 8. Comparison of repeatability in searching performance over various chromosome lengths (lower better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

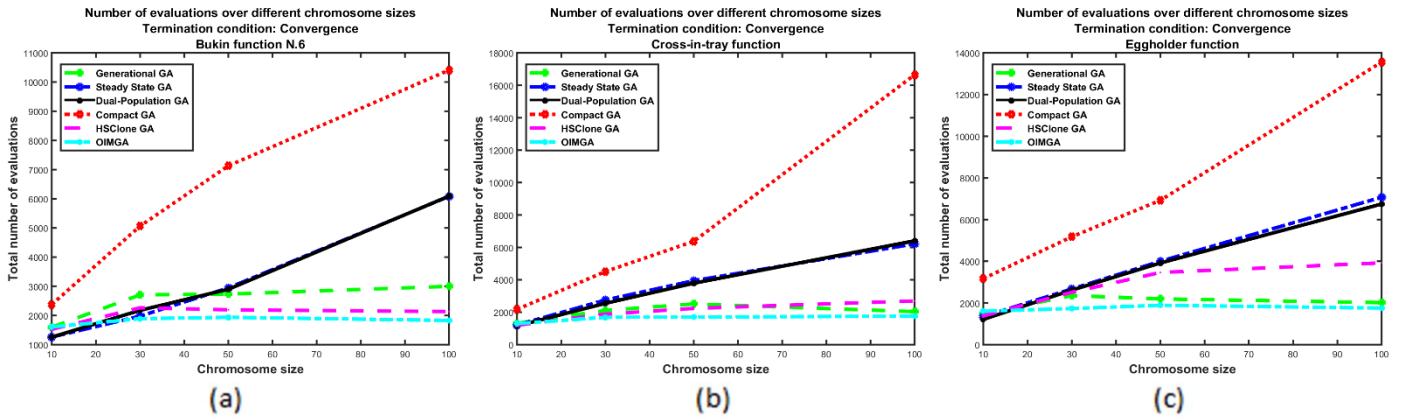


Figure 9. Comparison of convergence speed over different chromosome lengths (lower better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

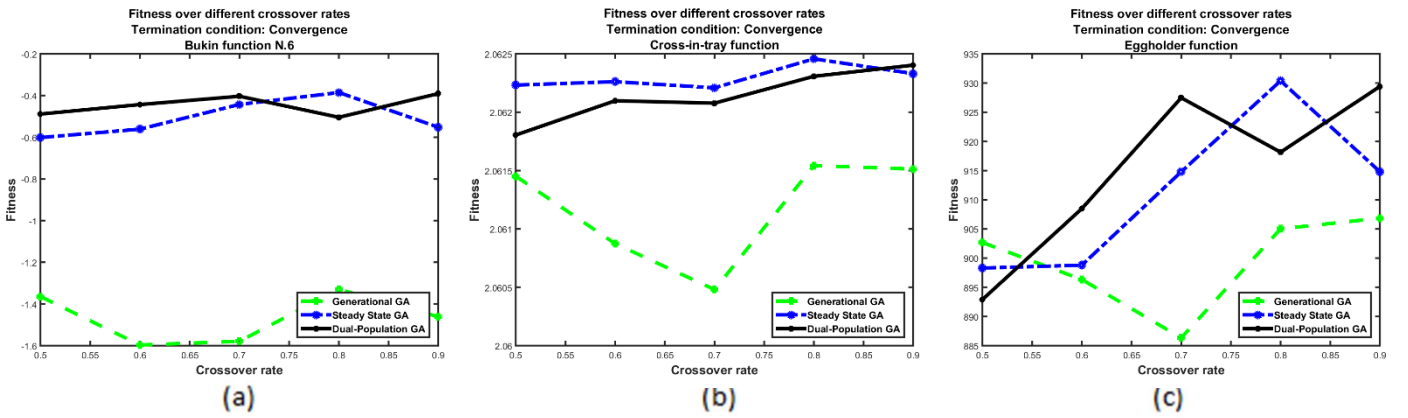


Figure 10. Comparison of searching performances over different crossover rates (higher better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

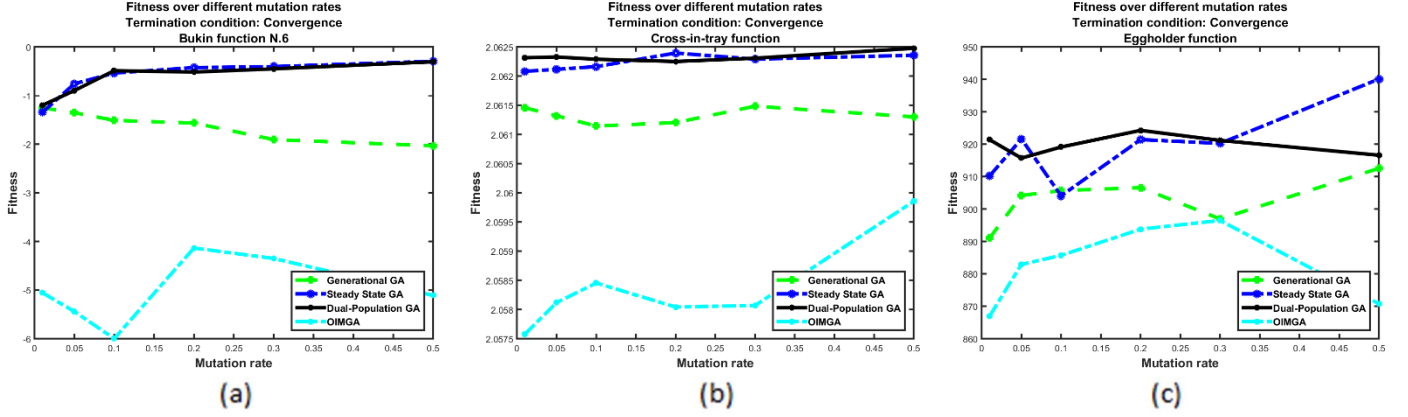


Figure 11. Comparison of searching performances over various mutation rates (higher better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

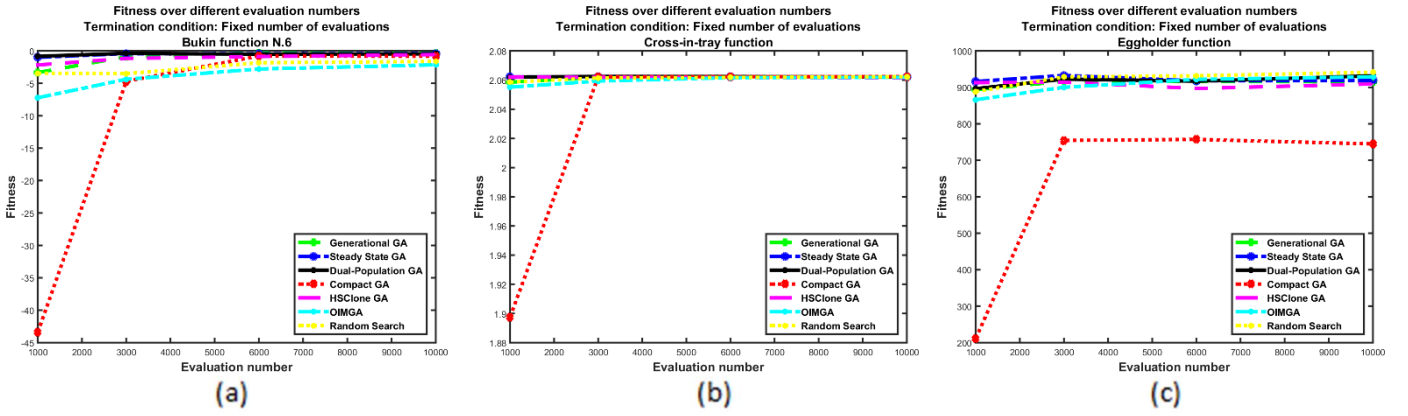


Figure 12. Comparison of searching performances over various fixed evaluations as the termination condition (higher better). Test functions: (a) Bukin N.6, (b) Cross-in-tray, (c) Eggholder.

By looking at the presented results, it can be inferred that the proposed method actually is one of the bests. In most of the situations, it achieved better results than the hardware-oriented GAs. It can be deduced from Fig. 6, Fig. 9, and Fig. 12 that dual-population scheme is more suitable for fix-evaluation (or fix-iteration) termination conditions. This means, it efficiently locates optimum solutions, but converges slowly to stop when a convergence-based termination condition is managing the run-time.

In addition, regarding the searching abilities and convergence properties, our implementation does not show statistically significant deviation from the single population SSGA, because the null hypothesis cannot be rejected in this case. This serves the purpose of increasing speed on hardware without remarkably deviating from a well-known and established GA, like SSGA.

## V. CONCLUSIONS

We investigated and evaluated a pipelined hardware implementation of steady state genetic algorithm with two populations that could reach the 2X theoretical speedup.

Our approach delivered performance and reliability that was statistically similar to the well-studied single population steady state genetic algorithm, while proving superior than the current hardware-oriented genetic algorithms. In our examples from a previously designed genetic algorithm processor, dual-population scheme achieved speedups up to 45% with the average of 34% over steady state genetic algorithm. Speedups are even larger over a canonical GA implementation. However, two populations double the memory footprint. Rapidly decreasing silicon area costs make the tradeoff between memory and computing speed skew towards computing speed for embedded GA implementations and delineate the potential of our approach.

## REFERENCES

- [1] S. P. H. Alinodhi, S. Moshfe, M. S. Zaeimian, A. Khoei, and K. Hadidi, "High-speed general purpose genetic algorithm processor," *IEEE Transactions on Cybernetics*, vol. 46, no. 7, pp. 1551-1565, July 2016.
- [2] K. Kobayashi, N. Yoshida, and S. Narazaki, "GAP/D: VLSI hardware for parallel and adaptive distributed genetic algorithms," *International Joint Conference on Computational Sciences and Optimization*, pp. 95-98, 2009.
- [3] P. Y. Chen, R. D. Chen, Y. P. Chang, L. S. Shieh, and H. A. Malki, "Hardware implementation for a genetic algorithm," *IEEE Transactions*



- on Instrumentation and Measurement, vol. 57, no. 4, pp. 699-705, April 2008.
- [4] M. S. Ben Ameer, A. Sakly, and A. Mtibaa, "Implementation of real coded genetic algorithms using FPGA technology," Tenth International Multi-Conference on Systems, Signals & Devices, pp. 1-6, Hammamet, Tunisia, 2013.
  - [5] 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.
  - [6] J. Kok, L. F. Gonzalez, and N. Kelson, "FPGA implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning," IEEE Transactions on Evolutionary Computation, vol. 17, no. 2, pp. 272-281, Apr. 2013.
  - [7] 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, Sept. 2013.
  - [8] 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 Transactions on Evolutionary Computation, vol. 14, no. 1, pp. 133-149, Feb. 2010.
  - [9] N. Nedjah, L. D. M. Mourelle, "An efficient problem independent hardware implementation of genetic algorithms," Neurocomputing, vol. 71, no. 1-3, pp. 88-94, Dec. 2007.
  - [10] O. Kitaura et al., "A custom computing machine for genetic algorithms without pipeline stalls," IEEE International Conference on Systems, Man, and Cybernetics, pp. 577-584, Tokyo, Japan, October 1999.
  - [11] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," IEEE Transactions on Evolutionary Computation, vol. 3, no. 4, pp. 287-297, Nov. 1999.
  - [12] Z. Zhu, D. J. Mulvaney, and V. A. Chouliaras, "Hardware implementation of a novel genetic algorithm," Neurocomputing, vol. 71, no. 1-3, pp. 95-106, Dec. 2007.
  - [13] M. S. Sharawi, J. Quinlan, and H. S. Abdel-Aty-Zohdy, "A hardware implementation of genetic algorithms for measurement characterization," 9th International Conference on Electronics, Circuits, and Systems, pp. 1267-1270, Sept. 2002.