International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:1, No:12, 2007

A Novel Genetic Algorithm Designed for Hardware Implementation

Zhenhuan Zhu, David Mulvaney, and Vassilios Chouliaras

***Abstract*—**A new genetic algorithm, termed the 'optimum individual monogenetic genetic algorithm' (OIMGA), is presented whose properties have been deliberately designed to be well suited to hardware implementation. Specific design criteria were to ensure fast access to the individuals in the population, to keep the required silicon area for hardware implementation to a minimum and to incorporate flexibility in the structure for the targeting of a range of applications. The first two criteria are met by retaining only the current optimum individual, thereby guaranteeing a small memory requirement that can easily be stored in fast on-chip memory. Also, OIMGA can be easily reconfigured to allow the investigation of problems that normally warrant either large GA populations or individuals many genes in length. Local convergence is achieved in OIMGA by retaining elite individuals, while population diversity is ensured by continually searching for the best individuals in fresh regions of the search space. The results given in this paper demonstrate that both the performance of OIMGA and its convergence time are superior to those of a range of existing hardware GA implementations.

***Keywords*—**Genetic algorithms, genetic hardware, machine learning.

I. Introduction

TO incorporate real-time learned intelligent functionality into applications such as fault tolerance, pattern recognition and optimal control, one approach is to embed machine learning in system-on-chip (SoC) implementations. Desirable features of such solutions are realization in a small silicon area to leave space for other SoC components, a flexible re-configuration structure to suit the needs of a wide range of applications, as well as short execution (learning) time. Desirable properties for hardware GAs are fast access to the individuals in the population, efficient use of available silicon area and flexibility in its structure to allow for a range of applications to be targeted, as well as good convergence performance, both in terms of calculation time and the quality of the result produced. The novel OIMGA approach presented in this paper has been specifically developed to ensure that it incorporates all of these properties. In particular, by keeping only the single current optimum individual, it addresses the memory and silicon area requirements. In addition, OIMGA is flexible in its structure in that it requires minimal changes to alter the population size to adapt to the requirements of a wide range of applications. Moreover, the results shown in this paper demonstrate that its rate of convergence is significantly faster than those of existing GA hardware methods described below.

A number of authors have described GA hardware solutions and applied these to embedded applications, either simply for algorithm acceleration [3][4][5], or for specific applications [1][2]. In hardware implementations, the need to provide substantial on-chip memory to store the population is generally recognized to affect adversely both the execution speed and the physical silicon area required. For example, in the roulette GA [6], memory is required to store the entire population data as well as the fitness values. Such memory could be provided on-chip, in which case it is likely to operate at full clock speed, but occupy significant physical area that could otherwise have been used for additional GAs or processing elements relevant to the solution of the problem at hand. The alternative is to provide off-chip memory, in which case not only may cost considerations dictate the use of slower memory requiring a number of clock cycles to access, but also, if a number of GAs are combined in a single device, it is unlikely that the data bandwidth will be sufficient to allow all GAs simultaneous access to their respective populations. The compact GA [3] is one approach specifically targetted to address this memory bottleneck issue. To permit their use in a wide range of applications, implementations of GAs need to be flexible in their structure to allow variations in the population size and the lengths of individuals [3][4]. For example, a suitable length for the individuals is typically influenced by the size of the solution space and the diversity of the population is often related to the number of individuals in that population. One important measure of the performance of GAs is their rate of convergence. A hardware implementation based on a 'half-siblings-and-a-clone' [1] was shown to shorten the GA convergence time.

Initially, we considered a large number of candidate GA algorithms with respect to their suitability for hardware implementation. Many conventional GA were discarded for the purposes of this study as not considered as they did not have any of the desirable properties for hardware

Manuscript received May 24, 2006.

Z. Zhu is with the Department of Electronic and Electrical Engineering, Loughborough University, Loughborough, LE11 3TU, UK.

D. Mulvaney is with the Department of Electronic and Electrical Engineering, Loughborough University, Loughborough, LE11 3TU, UK (phone: +44 1509 227042; fax: +44 1509 227014; e-mail: d.j.mulvaney@lboro.ac.uk).

V. Chouliaras is with the Department of Electronic and Electrical Engineering, Loughborough University, Loughborough, LE11 3TU, UK.

implementation. The implementations and preliminary results for the exiting hardware GAs we considered suitable for further investigation are described in section 2, section 3 then considers the new OIMGA solution that is designed to overcome the drawbacks of existing GAs. The results of the implementations of all the GAs in terms quality of results and calculation time are presented in section 4.

The contributions of this paper are: (a) the investigation of a number of existing GA solutions specifically targeted for embedded (though not necessarily hardware) implementations; (b) the introduction of a new GA with attributes specifically suited to hardware implementation; and (c) results that demonstrate the particular advantages of the new approach.

II. Investigation of Existing Hardware GAs

This section describes previous work that has developed GA approaches relevant to an efficient hardware realization. In section 4, these GAs, as well as OIMGA, are applied to benchmark problems to assess a number of aspects of their performance relevant to hardware implementation. Note that a modification to one of the existing approaches (the roulette GA) was made by the authors in order to reduce its memory usage and calculation times.

A. Hsclone GA

The 'Hsclone' GA was developed as a time-efficient approach based on the 'half-siblings-and-a-clone' crossover technique that manages the assignment of fitness probabilities to chromosomes [4]. A predefined fitness criterion is applied to the population and on the input data measurement. Based on the results obtained, the chromosomes that best meet the fitness criteria are kept, while others are replaced by the individuals that result from crossover should their fitness exceed a given threshold, or otherwise by a new randomly generated chromosome. In the authors' simulation, the threshold was the defined to be the error voltage equal to one-fourth of the maximum. Fig. 1 shows the psuedocode of the Hsclone GA, designed according to the description in [4].

In this GA, the crossover rate is normally changed based on the results obtained following each generation; where an improvement over the previous best solution lowers the crossover rate. The mutation rate is set by a threshold, where the larger its value the greater the number of chromosomes that are selected. The GA requires memory for storing its population and the selection of appropriate parameters for a particular application of the GA need to be determined empirically.

B. Roulette GA

Ramamurthy and Vasanth [6] describe a roulette wheel for crossover selection that is constructed as follows. The data element with the smallest error value is given highest rank and appears n times in the wheel, the member with the second smallest error value is given second rank and is duplicated n-1 times, and so on. The resultant size of the wheel is then $n(n+1)/2$, where n is the number of individuals in the population. Mutation is implemented by inverting one bit selected at random from an individual in the population that has also been selected at random. Following sorting in terms of fitness values, the n best individuals are kept for the subsequent generation. Fig. 2 shows the psuedocode of the Roulette GA.

```
l : length of individual
n : size of population
max_gen : maximum number of generations

population=randCreateIndvs(n, l);
best_indv= population (1);
best_fit=fitness(best_indv);

gn= max_gen;
while gn>0
  worst_fit=best_fit/4;   % set the crossover threshold
  indv_ptr=n;
  while indv_ptr >0
    fit_val=fitness(population(indv_ptr));
    if fit_val>best_fit
      best_indv= population(indv_ptr);
      best_fit= fit_val;
    else
      if  fit_val < worst_fit
        population(indv_ptr)=crossover(best_indv, randIndv());
      else
        population(indv_ptr)=crossover(best_indv,
                                       population(indv_ptr));
      endif
    endif
    indv_ptr = indv_ptr -1;
  endwhile
  gn=gn-1;
endwhile
```

Fig. 1 Psuedocode of the half-sibling-and-a-clone GA

In the implementation described in [6], it is evident that in addition to the memory was used for storing the individuals, their indices and their fitness values, an additional $n(n+1)/2$ memory units are needed for the roulette wheel during the selection of individuals for crossover. In addition, the calculation time is also adversely affected by the need to sort the population In order to mitigate these drawbacks, the authors developed an alternative roulette algorithm, shown in Fig. 3. Although this implementation is able to significantly reduce the memory requirement by removing the need to store the data associated with the roulette wheel, floating point calculations of the probabilities are now needed, and its hardware implementation will occupy significant silicon area. However, in most practical implementations, it is likely that a scaled integer calculation of the probabilities would be suitable or that floating point units would already exist in the hardware as part of the fitness calculations. In addition, the use of probabilities for selection is able to remove the need for sorting the population.

```
l :  length of individual
n : size of population
max_gen : maximum number of generations
m_rate : mutation rate
c_rate : crossover rate

r_wheel[1..(1+n)n/2]=createWheel (n);
population=randCreateIndvs(n);
fit_val[]=fitness(population);
indv_idx[]={1..n};
[fit_val, indv_idx]=sorting(fit_val, indv_idx);
best_fit=fit_val(1);
best_indv=population(indv_idx(1));

gn=max_gen;
while gn>0
  for i=1to n/2
    [p1, p2]=roulette(r_wheel, indv_idx);
    if (rand<c_rate)
      [children(i×2-1), children(i×2)]=crossover(population(p1),
                                                 population( p2));
    else
      [children(i×2-1), children(i×2,1)]=[ population(p1),
                                           population(p2)];
    endif
  endfor
  population=mutation(children, m_rate, n);
  fit_val[]=fitness(population);
  indv_idx[]={1..n};
  [fit_val,  indv_idx]=sorting(fit_val, indv_idx);
  if  best_fit<fit_val(1)
    best_fit= fit_val(1);
    best_indv=population(indv_inx(1));
  else
     population(indv_idx(n))=best_indv;
     indv_idx= rightShift(indv_idx);
  endif
  gn=gn-1;
endwhile
```

Fig. 2 Pseudocode for the roulette GA

C. Compact GA

The parameters for the compact GA [3] are the population size n and the chromosome length l. A population is represented by an l-dimensional probability vector p, where the $p[i]$ are the probabilities that the i^{th} bit in an individual, randomly selected from the population, will have the value of unity. Initially, all the values in p are set to 0.5 and two individuals, a and b, have their constituent bits initialized according to p and their corresponding fitness values, f_a and f_b, are determined. The operation of the GA then involves repeatedly comparing the fitness values and, according to its outcome, then updating the individuals. If $f_a \geq f_b$ then the probability vector will be updated towards the individual a, otherwise towards b. If $a[i] = 1$ and $b[i] = 0$ then $p[i]$ is incremented by $1/n$ and conversely if $a[i]=0$ and $b[i]= 1$, $p[i]$ is decremented by $1/n$. The cycle halts once each entry in p is either zero or unity, at which point p holds the final solution. Fig. 4 shows the psuedocode of the compact GA.

The compact GA is straightforward to extend in terms of size of population and the length of chromosome. There is no need for memory to store the population or fitness, and the circuitry is simplified by the absence of crossover. However, the practical results presented later in the paper indicate that the compact GA has relatively long calculation times.

```
l : length of individual
n : size of population
max_gen : maximum number of generations
m_rate : mutation rate
c_rate : crossover rate

population=randCreateIndvs(n, l);
fit_val[]=fitness(population);
best_fit=0;
[best_indv, best_fit]=getBest(population, best_fit);
 prob[]=calProb(fit_val);

 gn=max_gen;
 while gn>0
   for i=1 to n/2
     [p1, p2]=roulette(prob);
     if (rand()<c_rate)
       [children(i×2-1),  children(i×2)]= crossover(population(p1),
                                                    population( p2));
     else
       [children(i×2-1), children(i×2,1)]=[ population(p1), population(
                                            p2)];
     endif
    endfor
  population=mutation(children, m_rate, n);
  fit_val[]=fitness(population);
  [best_indv, best_fit]=getBest(population, best_fit);
  bad_index=getWorst(population);
  [population(bad_index), fit_val(bad_index)]=[best_indv, best_fit];
  prob=calProb(fit_val);
  gn=gn-1;
endwhile
```

Fig. 3 An alterative algorithm for the roulette GA

```
n:  population size
l:  chromosome length

for i = 1 to l
  p[i] = 0.5;
endfor
repeat
  for i = 1 to l
    a[i]= 1 with probability p[i], 0 otherwise
    b[i]= 1 with probability p[i], 0 otherwise
  endfor

  fa= fitness(a);
  fb= fitness(b);
  for i = 1 to l
    if  fa  ≥ fb  then
      if a[i]= 1 and b[i]=0 then p[i]= min(1, p[i]+1/n); endif;
      if a[i]= 0 and b[i] = 1 then p[i]= max(0,p[i]-1/n); endif
    else
      if a[i] = 1 and b[i] = 0 then p[i]= max(0, p[i]-1/n); endif
      if a[i] = 0 and b[i] = 1 then p[i]= min(1, p[i]+1/n); endif
    endif
  endfor
until each p[i]∈{0,1}
```

Fig. 4 Psuedocode for the compact GA

III. OIMGA Algorithm

OIMGA incorporates two searches that interact hierarchically, namely a global search and a local search. In the global search, regions are selected sequentially from the entire search space for more detailed exploration by the local search. In the global search, a single individual is maintained (termed **topChrom**) that is the best (according to the fitness criterion) obtained from all the local searches carried out so far. The local search investigates the regions selected by the global search in order to determine the local optimum individual (LOI). This is achieved by generating an initial population in a narrow range using micro mutation. If the micro mutation results in a better individual this becomes the new LOI. The process is repeated until a termination criterion is satisfied.

As the best individual among all generations that have been investigated is always kept, then the proof given by Radolph [7] can be applied directly to demonstrate that OIMGA is convergent. As the algorithm repeatedly initializes the population space following a global search, OIMGA is very effective in maintaining diversity and preventing premature convergence.

Compared with the existing methods described above, the convergence time of OIMGA is likely to be shorter due to reductions both in the total search space explored and in the population size [8]. A further execution speed enhancement in the hardware implementation is also easily identifiable since the executions of the global and the local searches are prime candidates for hardware pipelining. Table I shows the parameters available to a designer using the OIMGA algorithm, while Fig. 5 shows the pseudocode of OIMGA itself.

TABLE I
OIMGA Parameters

l	individual length
n	population size
m	the size of the miniature space around the LOI
t_gens	maximum number of consecutive global generations without improvement
k_gens	maximum number of consecutive local generations without improvement
d_adjustor	range of mutation of an individual
m_rate	probability of mutation

IV. OIMGA Hardware Design

Fig. 6 shows the main structure of the hardware implementation of OIMGA. The **LOI-generator** initiates the local process by randomly producing a population that includes *n* individuals, and then searches for the LOI. In the **micro-mutation unit,** the individuals are allowed to evolve in value only within the range indicated by the value of **d_adjustor** and any change of range is controlled by the **d_controler.** The fitness value of the generated individual is calculated by the **fitness-unit** and the **local-evaluator** compares the fitness of the current LOI with that of the previous one and replaces it if its fitness is better. The search in the local space is repeated *m* times. If, during these searches, a new LOI is not found then the range that **d_adjustor** indicates is decreased. Should the fitness of the LOI not improve over *k_gens* cycles, then the LOI is sent to the **global evaluator**. The global evaluator implements the global process and retains the globally best individual and its fitness value found from all the local searches. The global process terminates when the fitness has not improved over *t_gens* operations of the local process.

```
g=t_gens;
while g>0 % start a global search
 d=d_adjustor;
 for i=1 to :n
   loiChrom= randCreateIndv(l);  % random l-bit individual
   loiFit=fitness(loiChrom,l);   % find its fitness
   if loiFit>bestFit          % keep an elite individual and
    bestChrom=loiChrom;   % its fitness
    bestFit=loiFit;
   endif
 endfor

 k=k_gens;
 while k>0     % start a local search
  update=0;   % number of updates of tempChrom and bestChrom
  for i=1 to :m      % perform local search m times
   tempChrom=bestChrom;
   for j=d to l     % produce a micro mutation in the range d to l
    if rand()<m_rate      % 'rand' is a random number
      tempChrom(j)=not(tempChrom(j));     % invert the jth bit
    endif
   endfor
   tempFit=fitness(tempChrom,l);
   if tempFit>bestFit
    bestChrom=tempChrom;    % keep the elite local individual
    bestFit=tempFit;              % and its fitness
    update=update+1;
    k=k_gens;
    d=d_adjustor;
   endif
  endfor
  if update=0         % decrease range (d-l) if no update
   d=d+1;
  endif
  k=k-1;
 endwhile
 if bestFit>topFit              % keep elite global individual
  topChrom=bestChrom;    % and its fitness
  topFit=bestFit;
  g=t_gens;
 endif
 g=g-1;
endwhile
```

Fig. 5 The pseudocode for the OIMGA algorithm

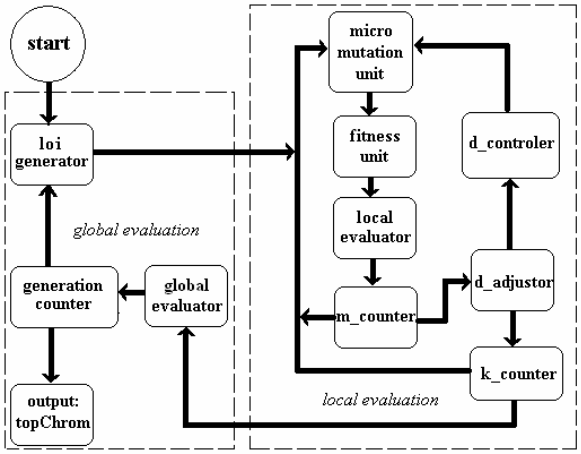


Fig. 6 The main structure of OIMGA

A. LOI Generator

The LOI generator shown in Fig. 7 includes a random number generator **RNG** that produces an *l*-bit random individual whose fitness value is calculated by the **fitness unit** and stored in the register **loiFit**. The unit **cmp1** is used to compare the fitness of **loiFit** with that of the best fitness value held in **bestFit** and, if it is better, **bestFit** is replaced by **loiFit** and the new individual (of length *l*) replaces that held in the register **bestChrom**. The **n bit counter** ensures that the entire process is carried out *n* times, where *n* is the population size. Note that in order to modify the size of the population, it is only necessary to change the length of the counter.

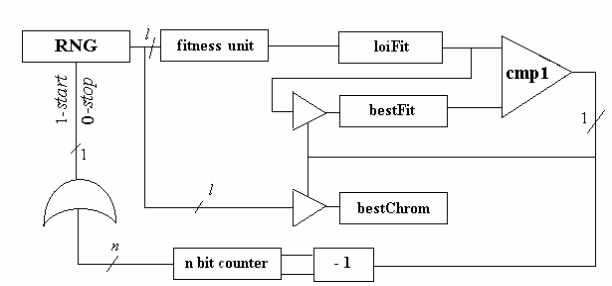


Fig. 7 LOI generator

B. Micro-Mutation Unit

The micro-mutation unit is shown in Fig. 8. If the probability of mutation **pm** is greater than **RNG***i* and **d_MRSR***i* is set, the i^{th} bit of **bestChrom** is mutated. The register **tempChrom** holds the value of the chromosome following mutation and is evaluated in the **fitness unit**. If its fitness is better than that in **bestFit** (as determined in the **local evaluator** shown in Fig. 9), the signal **cmp2** operates the tri-state gate to replace **bestChrom** by the value in **tempChrom**. To modify the length of the individual, a corresponding change can be made to the number of bits in the micro mutation unit.

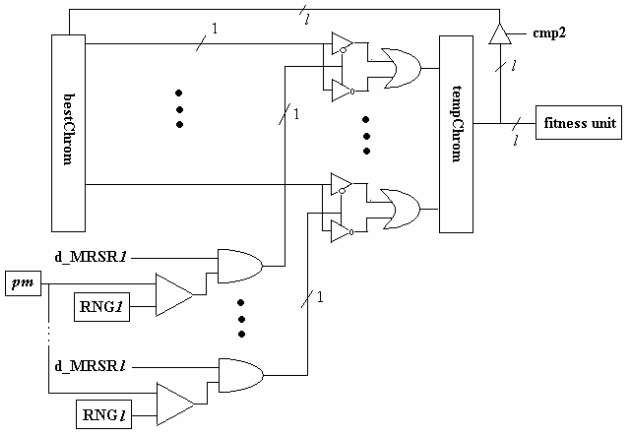


Fig. 8 Micro mutation unit

C. Local Evaluator

The local evaluator shown in Fig. 9 uses the fitness values to select the better individual from **tempFit** and **bestFit**, and keeps this elite individual and its fitness value during local evolution.

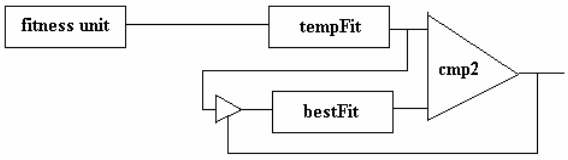


Fig. 9 Local evaluator

D. Adjusting the Range of Mutation

During an evolution process, generally the times between modifications to the fitness values decrease, indicating that the evolution is converging to a final value. To speed up convergence, it is appropriate to reduce the allowed change of mutation values in order to investigate the space in the more immediate vicinity of the current best individual.

Initially, the bits in the **mask right shift register** shown in Fig. 10 are all set, $MRSR_i=1$, $i \in [1,l]$. The initial value of the range held in **d_initial** is set to a predefined value, signifying that all but this number of bits in the individuals should be mutated. This value is copied into **d_counter**. The update register (**update**) is initialized with 0 and is incremented whenever the local evaluator replaces the current best individual. The value in **d_counter** defines the number of shifts that are performed by the MRSR (with the left-most bit zero filled); at each shift **d_counter** decrements by 1. To understand the operation, consider the case where the initial value held in **d_counter** is 3. In this case, following the shift operations, the state of MRSR is shown as follows.

$$d\_MRSR_i = \begin{cases} 0 & 0 \le i \le 3 \\ 1 & 4 \le i \le l \end{cases} \quad (1)$$

These values indicate that the range of mutation is in [4, *l*]. During local evolution based on LOI, **update** will be increased by 1 if **bestChrom** and **bestFit** are replaced. After each generation of local evolution, **d_initial** will increase by 1 if **update** is still 0, thereby reducing the number of bits that are mutated in the micromutation unit.

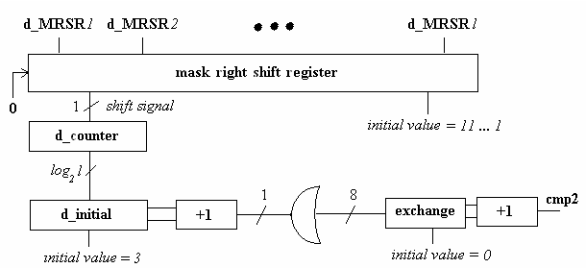


Fig. 10 Circuitry to adjust the range of mutation

E. Global Evaluator

The principle of operation of the **global evaluator**, shown in Fig. 11, is very similar to that of the local evaluator. The global evaluator selects the better individual from **bestFit** and **topFit**, and keeps the elite individual from all generations and its corresponding fitness value in **topChrom** and **topFit** respectively.

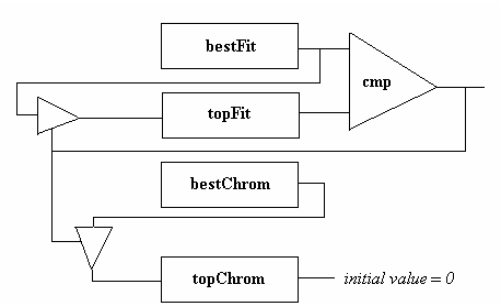


Fig. 11 Global evaluator

V. Results

To evaluate the efficiency of a number of hardware implementations of GAs with OIMGA, namely half-siblings-and-a-clone [1], roulette [6] and compact GA [3], the two benchmark functions defined by Zhang and Zhang [9] shown in the following equations were used.

$$f_1(x) = |(1-x)x^2 \sin(200\pi x)| \qquad x \in (0,1) \qquad (2)$$

$$f_2(x) = (1-2\sin^{20}(3\pi x) + \sin^{20}(20\pi x))^{20} \qquad x \in (0,1) \qquad (3)$$

$f_1(x)$ has 200 local maximum and minimum values in its defined range (Fig. 12), while $f_2(x)$ has 20 local maximum and minimum values in its defined range (Fig. 13). It is very difficult to determine analytically the maximum and minimum values of the two functions by methods other than using some form of search [9].

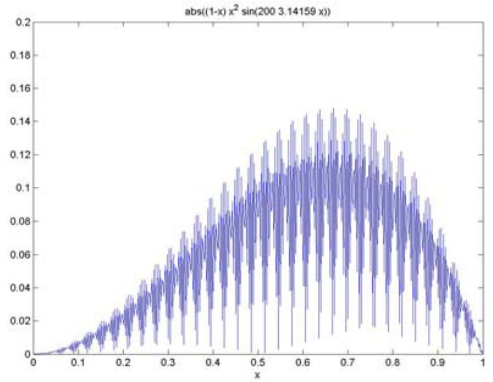


Fig. 12 Benchmark function $f_1(x)$

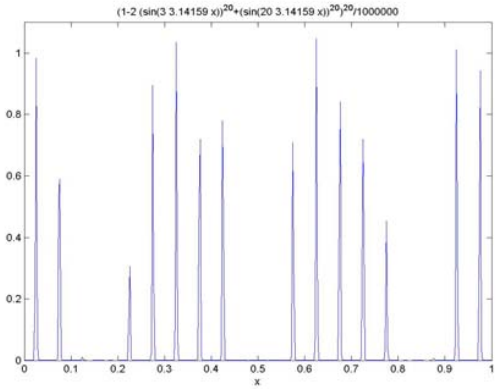
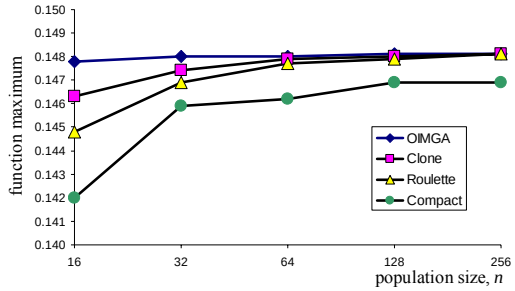


Fig. 13 Benchmark function $f_2(x)$

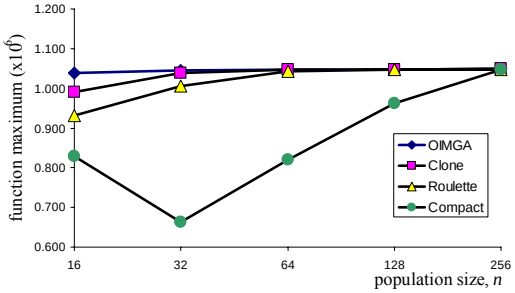
Presented here are results of a number of experiments to assess the following three aspects of the four hardware GA implementations: the quality of the solution produced, the calculation time and the hardware component requirements.

The GA implementations (other than OIMGA) were carried out according to the descriptions given by the respective authors. The simulations were all developed and run in MATLAB [10] on the same host computer system. Since MATLAB cannot fully reproduce the cycle-accurate timings of a hardware implementation, the timings can only be regarded as indicative.

In the first set of experiments, the performance of the GAs in determining the maximum values of the functions $f_1(x)$ and $f_2(x)$ were investigated for various values of population size and individual lengths. Fig. 14 shows that for a fixed individual length, OIMGA outperformed the other GA implementations, particularly for small populations. The performance of the compact GA was noticeably inferior to the other implementations. The poor performance of the compact GA was also apparent when the population size was fixed and the maximum function values determined for a range of lengths of the individuals, Fig. 16. The remaining three GAs all performed similarly under this test.

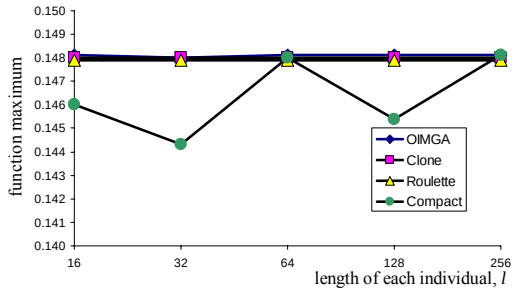


(a) Estimated $f_1(x)$ maxima

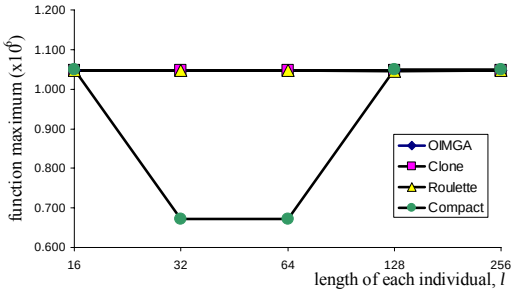


(b) Estimated $f_2(x)$ maxima

Fig. 14 Maxima of the benchmark functions found by the GAs for a range of population sizes and at a fixed individual length (l) of 32. Each data point shown was calculated from results averaged over 200 tests, except for the compact GA where only 20 tests were carried out.



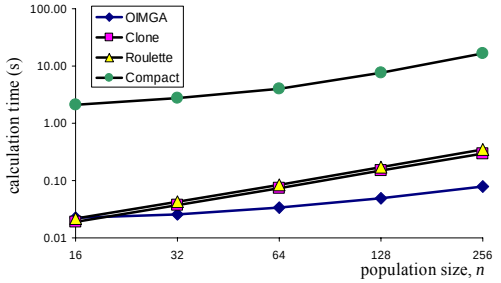
(a) Estimated $f_1(x)$ maxima



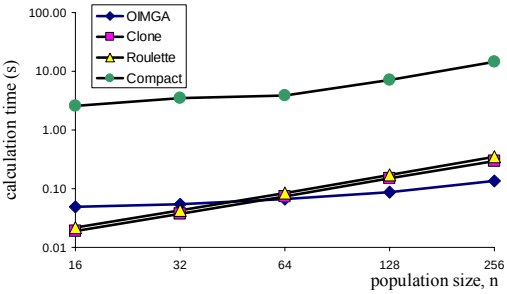
(b) Estimated $f_2(x)$ maxima

Fig. 15 Maxima of the benchmark functions found by the GAs for a range of individual lengths and at a fixed population size (n) of 128. Each data point shown was calculated from results averaged over 200 tests, except for the compact GA where only one test was carried out.

The second set of experiments investigated the calculation times to reach convergence when determining the maximum values of the functions $f_1(x)$ and $f_2(x)$ for the different population sizes and individual lengths. In Fig. 16, it can be seen that the compact GA performed poorly across a range of population sizes, with the calculation times often being two orders of magnitude greater than those of the other GAs. It can be seen from Fig. 16 that, as the population size is increased, the calculation times of OIMGA increase less steeply than those of the other GAs. More detailed investigations revealed that, with the doubling of the population size, the calculation times for OIMGA increased at only half the rate of the half-siblings-and-a-clone and the roulette GAs. Fig. 17 shows that the calculation times for the compact GA were particularly long when the length of the individuals was increased beyond 32. These results also show that the other GA methods produced shorter calculation times and OIMGA performed particularly well in the more demanding cases where the individuals were of greater length and the population larger.



(a) Estimated $f_1(x)$ calculation times



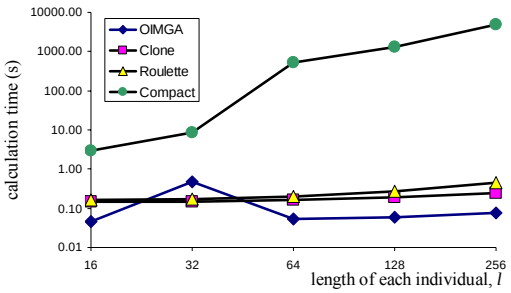
(b) Estimated $f_2(x)$ calculation times

Fig. 16 Calculation times of the benchmark functions found by the GAs for a range of population sizes and at a fixed individual length (l) of 32. Each data point shown was calculated from results averaged over 200 tests, except for the compact GA where only 20 tests were carried out.

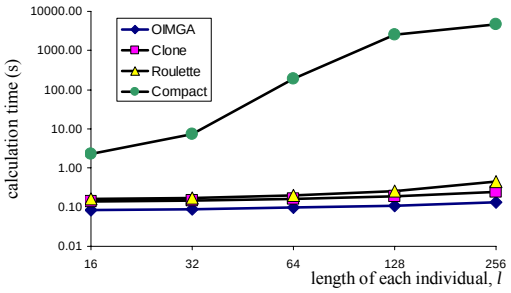
In order to generate representative figures, the experimental procedure to produce the results involved adjustment of the respective parameters of each of the GAs (other than for l and m whose values were purposely varied to obtain the results). The parameters used by OIMGA for the estimation of the maximum values of $f_1(x)$ and $f_2(x)$ are given in Table II. Note that altering the width of the fitness value affects not only the

system performance, but also has an effect on other hardware requirements, such as the width of the comparator.

Hardware implementations of GAs mainly consist of random number generators, comparators, registers and memory. The requirement of each component can be described with its total bit number (TBN). For example, if there are ten 8-bit registers in a circuit, their TBN is 80 bits. To illustrate the relative complexities of the GAs investigated in the current work, the values in Table III were obtained from algorithmic estimates of the hardware requirements of four different classes of component. It can be seen that the TBN for the compact GA and OIMGA solutions are an order of magnitude less than those for the other GA methods. However, in contrast with OIMGA, the modest hardware requirement of the compact GA has clearly been achieved at the expense of performance.



(a) Estimated $f_1(x)$ calculation times



(b) Estimated $f_2(x)$ calculation times

Fig. 17 Calculation times of the benchmark functions found by the GAs for a range of individual lengths and at a fixed population size (n) of 128. Each data point shown was calculated from results averaged over 200 tests, except for the compact GA where only one test was carried out

TABLE II
OIMGA PARAMETER VALUES

function	*m*	*t_gens*	*k_gens*	*d_adjuster*	*pm*	width of fitness value
$f_1(x)$	10	4	5	3	0.382	32
$f_2(x)$	16	6	5	4	0.382	32

TABLE III
HARDWARE REQUIREMENTS OF THE GA IMPLEMENTATIONS

GA	random number generators	comparators	registers	memory	total
OIMGA	160	224	296	0	680
Clone	32	96	256	4096	4480
Roulette	59	64	478	8192	8793
Compact	256	262	352	0	870

VI. CONCLUSION

The paper has introduced a new GA algorithm that is particularly suited for hardware implementation because of its minimal memory requirement and its ability to allow both the size of the population and the length of the individuals to be altered simply by replicating existing logic units. When run on benchmark problems, the new algorithm compared favorably with other hardware solutions found in the literature, both in terms of its execution time and in its performance on benchmark problems. Future publications will present the results of our investigations of implementing the GAs in a hardware design language and running cycle-accurate simulations in order to determine more precisely their relative performances.

REFERENCES

[1] Aporntewan, C. and Chongstitvatana, P., "A hardware implementation of the compact genetic algorithm", 2001 IEEE Congress on Evolutionary Computation, Seoul, Korea, 2001, pp.27-30.
[2] Wakabayashi, S., Koide, T., Toshine, N., Yamane, M. and Ueno, H., "Genetic algorithm accelerator GAA-II", *Proc*. Asia and South Pacific Design Automation Conference, Yokohama, Japan, January 2000.
[3] Scott, S.D., Samal, A. and Seth, S., "HGA: A hardware-based genetic algorithm", *Proc*. 3rd ACM/SIGDA Int. Symp. on FPGAs, 1995, pp.53-59.
[4] Sharawi, M.S., Quinlan, J. and Abdel-Aty-Zohdy, H.S., "A hardware implementation of genetic algorithms for measurement characterization", IEEE 9th International Conference of Electronics, Circuits, and Systems, Dubrovnik, Croatia, 3, 2002, pp.1267-1270.
[5] Hauser, J.W. and Purdy, C.N., "Sensor data processing using genetic algorithms", IEEE Mid- West Symp. on Circuits and Systems, August 2000.
[6] Ramamurthy, P. and Vasanth, J., "VLSI implementation of genetic algorithms" (under review).
[7] Radolph, G., "Convergence analysis of canonical genetic algorithms", *IEEE Trans. Neural Networks*, 5(1), 1994, pp.96-101.
[8] Li, J. and Wang, S., "Optimum family genetic algorithm", *Journal of Xi'an Jiao Tong University*, 38, Jan 2004.
[9] Zhang, L. and Zhang, B., "Research on the mechanism of genetic algorithms", *Journal of Software*, 11(7), 2000, pp.945-952.
[10] Matlab, http://www.mathworks.com/