View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NAIST Academic Repository

A Hardware Implementation Method of Multi-Objective Genetic Algorithms

Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata†, Keiichi Yasumoto and Minoru Ito
Graduate School of Information Science,
Nara Institute of Science and Technology.
Ikoma, Nara 630-0192, Japan
{tatsu-ta,yosihi-m,yasumoto,ito}@is.naist.jp

† Department of Information Processing and
Management, Shiga University.
Hikone, Shiga 522-8522, Japan
shibata@biwako.shiga-u.ac.jp

***Abstract*— Multi-Objective Genetic Algorithms (MOGAs) are approximation techniques to solve multi-objective optimization problems. Since MOGAs search a wide variety of pareto optimal solutions at the same time, MOGAs require large computation power. In order to solve practical sizes of the multi objective optimization problems, it is desirable to design and develop a hardware implementation method for MOGAs with high search efficiency and calculation speed. In this paper, we propose a new method to easily implement MOGAs as high performance hardware circuits. In the proposed method, we adopt simple Minimal Generation Gap (MGG) model as the generation model, because it is easy to be pipelined. In order to preserve diversity of individuals, we need a special selection mechanism such as the niching method which takes large computation time to repeatedly compare superiority among all individuals in the population. In the proposed method, we developed a new selection mechanism which greatly reduces the number of comparisons among individuals, keeping diversity of individuals. Our method also includes a parallel execution architecture based on Island GA which is scalable to the number of concurrent pipelines and effective to keep diversity of individuals. We applied our method to multi-objective Knapsack Problem. As a result, we confirmed that our method has higher search efficiency than existing method.**

I. Introduction

Multi-Objective Genetic Algorithms (MOGAs) are techniques to solve multi-objective optimization problems which optimize multiple conflicting objectives at the same time. Multi-objective optimization problems have a set of optimal solutions called Pareto optimal solutions which may include many solutions. So, MOGAs require large computation power to search pareto optimal solutions uniformly. For practical use of MOGAs, it is mandatory to solve the problems in short time. For this purpose, existing software implementation techniques are not enough. So, we need an efficient hardware implementation technique for MOGAs.

There are several existing studies regarding to hardware implementation of optimization algorithms. Ant Colony Optimization (ACO) and Simulated Annealing (SA) have been implemented on hardware platforms in [8], [9], respectively. Several techniques for hardware implementation of GAs have also been proposed. In [13], Wakabayshi et al. implemented a GA with an adaptive selection. In [11], Kobayashi et al. implemented a GA for extraction of disconnected closed loops on FPGA. In order to improve calculation speed of these circuit, it is important to prevent pipeline stall. Some of existing studies adopt steady-state GAs since they achieve less pipeline stalls than traditional generation models. In [10], hardware GA called H^3 engine that adopts steady-state GA was implemented. In [1], Barry et al. developed hardware circuits for Set Coverage Problem using Steady-state GA. In [2], Aporntewan et al. proposed a hardware implementation technique for Compact Genetic Algorithm on FPGAs. These existing studies target GAs with a single objective function. As long as we know, there is no study aiming at hardware implementation of MOGAs.

Several software implementation techniques for MOGAs have been proposed so far. In [3], Deb et al. proposed an algorithm called NSGA-II (Non-Dominated Sorting Genetic Algorithm). In [17], Zitzler et al. proposed an algorithm called SPEA-II (Strength Pareto Evolutionary Algorithm-II). These algorithms are difficult to be implemented as hardware circuits, since they use general generation models and complex selection mechanisms such as the niching method. So, we need an appropriate method for implementing MOGAs as hardware.

In this paper, we propose a new method to easily implement MOGAs as high performance hardware circuits. The proposed method is an extension of our previous method [15] which implements single objective GAs as hardware circuits. In the proposed method, we adopt simple Minimal Generation Gap (MGG) model as the generation model, because it is easy to be pipelined. In order to preserve diversity of individuals, we developed a new selection mechanism which greatly reduces the number of comparisons among individuals, keeping diversity of individuals. Our method also includes a parallel execution architecture based on Island GA which is scalable to the number of concurrent pipelines and effective to keep diversity of individuals.

We applied our method to multi-objective Knapsack Problem and compared search efficiency between our method and NSGA-II. Through these experiments, we confirmed that the proposed method has higher search efficiency than NSGA-II.

In the following Sect. II, we describe outline of our architectures for hardware implementation of single objective GA proposed in [15]. In Sect. III, we describe basic ideas of our proposed method. In Sect. IV, we describe details of our hardware implementation techniques. In Sect. V, we describe the experimental results. Finally, we conclude the paper in Sect. VI.

II. Hardware Implementation of Single Objective GA

In this section, we briefly explain our previously proposed method for hardware implementation of single objective GA on FPGA [15]. First, we describe the outline of hardware implementation of GA circuits in Sect. II-A. Then, we describe the generation model used in the architecture in Sect. II-B. Finally, we describe the technique for parallel execution in Sect. II-C.

A. Outline of Hardware Implementation of Single Objective GA

We proposed a general hardware architecture for single objective GA [15]. The goal of the architecture is to synthesize efficient hardware circuit of single objective GA for a given problem and problem size (size of each solution) which fully utilizes the target FPGA device. To achieve the goal, the proposed architecture has the following features. (1) It is easy to implement on FPGA and general enough to be applied to various problems. (2) It has good performance by fully utilizing target FPGA device and constructing parallel execution circuit of GA operations.

For the above (1), efficient memory utilization is essential. So we adopted a generation model based on Minimal Generation Gap (MGG) model [5]. The outline of this generation model is explained in Sect. II-B. In this architecture, hardware modules corresponding to GA operations such as crossover and mutation have to be designed separately, and the throughputs of these modules have to be the same so that they can be executed in a pipelined manner. This module-based design allows our architecture to be applied to various GA problems.

For the above (2), reducing synchronization among multiple parallel pipelines is important. We adopted Island GA (IGA) model [4] as the parallel execution architecture. This parallel architecture is explained in Sect. II-C.

B. Generation Model of Hardware GA

If the population management mechanism is implemented as a hardware circuit in a straight-forward way, extra memory to store newly generated individuals is required in addition to the memory for storing current population. In [1] and [2], survival-based steady-state GA and Compact Genetic Algorithm are used to reduce sizes of a hardware circuit and memory, respectively.

Survival-based steady-state GA replaces the individual with the worst fitness value in the current population by a newly generated individual with a better fitness value. Steady-state GA always keeps track of the worst individual. This requires extra clocks and makes pipelining difficult. Compact Genetic Algorithm does not retain a population. Instead, it retains a probability distribution to approximate the set of chromosomes in the current population. Compact GA assumes that all chromosomes are represented only by 0 and 1, and there is no straight-forward way to apply this algorithm to TSP or other practical problems.

In the proposed method, we use a simplified MGG model [5]. In this model, two individuals are picked up from the current population. Crossover and mutation operations are applied to these individuals to generate an offspring individual. This offspring individual is then evaluated. Selection operation selects the individual with the highest fitness value from the family (an offspring individual and the parent individuals) and replaces the worst individual in the family with it. This simplification makes it easy to construct pipelined and parallel circuit for processing individuals, and greatly reduces the required memory for storing individuals.

With this method, however, candidate solutions tend to converge towards one point in the search space since it targets single objective. We describe the enhanced technique to solve this problem in Sect. III-B. In Sect. III-B, we extend this method to be able to obtain wide variety of candidates solutions for multi-objective optimization problems.

C. Outline of Parallel Architecture

There are various techniques for parallel execution of GA. In our architecture, we use the technique of Island GA (IGA). IGA divides the population into several sets. Each set is regarded as an island, and population in each island evolves independently. Tiny fraction of the population periodically migrates to another island so that all islands cooperatively search for a good solution. Since IGA tends to retain better diversity of individuals than simple GA (SGA, hereafter), it hardly falls into a local optimum, and thus it has better search efficiency than SGA.

For parallel processing in a hardware circuit, maximum operating frequency may be decreased due to synchronization among parallel processing units. By using the IGA model for parallel execution of GA in a hardware circuit, each island (processing unit) is only required to periodically exchange individuals with its neighboring island for parallel execution. So, the synchronization mechanism can be very simple and its critical path does not depend on the number of parallel processing units. This greatly contributes to the performance scalability of the resulting circuits.

III. Proposed Method

In Sect. II-A, we described an architecture suitable to implement single objective GAs as hardware circuits. However, this architecture cannot be directly applied to MOGAs since it is designed to keep individuals with better fitness values calculated by a single objective function in the population at each generation. On the other hand, MOGAs need to keep a wide variety of individuals in the population at each generation to find (near) pareto optimal solutions uniformly. In order to keep diversity of individuals, existing MOGAs adopt the niching method [6]. However, it is difficult to implement those mechanisms as pipelined hardware modules, since they require repeated comparison among all individuals in the population. Therefore, if we implement MOGAs as hardware circuits in a straightforward way, this part would be bottleneck.

According to the above discussion, we propose a new selection mechanism to keep diversity of individuals which is suitable for hardware implementation. Our proposed mechanism can be implemented as pipelined modules on a hardware platform. Also, we propose an overall architecture suitable to implement MOGAs as hardware circuits. Our proposed architecture is scalable with respect to the number of concurrent pipelines, that is, within the available circuit size, the parallel degree can be increased with low overhead to improve search speed of MOGAs.

In the following subsections, first we define multi-objective optimization problems. Our selection mechanism for MOGAs is explained in Sect. III-B. Finally, we describe our architecture for parallel execution of MOGAs in Sect. III-C.

A. Multi-Objective optimization problem

Multi-objective optimization problems are problems to find solutions which optimize multiple conflicting objectives at the same time. So, the superior solution candidate cannot be determined by comparison of fitness values of a single objective.

Let I denote a set of objectives. Let x and y denote solution candidates (individuals). Let $f_i(x)$ denote the fitness value of individual x with respect to the objective $i \in I$. We say that x *is superior to* (or *dominates*) y if the following condition holds.

$$\exists_{i \in I}(f_i(x) > f_i(y)) \wedge \forall_{i \in I}(f_i(x) \geq f_i(y))$$

Pareto optimal solutions are solutions which are not dominated by all other solutions in the whole search space. In general, pareto optimal solutions are represented by a set of points in the search space and could contain many points. Multi-objective optimization problems are problems to find all of pareto optimal solutions.

MOGAs is one of the approximation techniques to find near pareto optimal solutions as uniformly as possible for multi-objective optimization problems.

B. Proposed mechanism for keeping diversity of individuals

We propose a selection operation to be easily implemented as a pipelined hardware module where individuals with similar chromosomes are removed prior to others to keep diversity in the population.

In our method, two individuals (parent individuals) are chosen from the current population (all solution candidates). Then crossover and mutation operations are applied to them to generate a new individual (offspring individual). The fitness value of each objective is calculated for the offspring individual. These operations can be implemented similarly to our method for single objective GA explained in Sect. II-B.

The selection operation is quite different from our previous method. In our method, two selection operations called *normal selection* and *biased selection* are used. Both of selection operations work as follows: (1) Two parent individuals are compared to investigate superiority between them. Then dominated parent individual is compared with the offspring individual. The offspring individual is replaced with the dominated parent individual in the current population, if they satisfy the condition explained below (This operation is called Replacement Part). (2) The offspring individual is compared with all candidate solutions in the current population. If there are individuals with the same chromosome as the offspring individual in the current population, they are removed from the population (This operation is called Overlap Rejection Part).

normal selection and *biased selection* differ in the policy of replacing the parent individual with the offspring individual. In Replacement Part, the condition of *normal selection* is whether the offspring individual dominates the parent offspring. The condition of the *biased selection* is whether the fitness value of the offspring individual is superior to that of the parent individual, with respect to a single specified objective.

In Overlap Rejection Part, the offspring individual is compared with all individuals in the current population in order to remove the individuals with the same chromosome. In our proposed method, however, we adopt a method to compare individuals by their fitness values. We will explain the reason in Sect. IV-E. If the solution candidates with the same fitness value as the offspring individual are found, they are marked. If it does not satisfy the condition of Replacement Part, the marked individuals are compared with the offspring individual. In that case, if the offspring individual dominates a marked individual, the marked individual is replaced with the offspring individual in the population.

C. Parallel Execution Architecture for Multi-Objective GA

Similarly to our previous method in Sect. II-C, we adopted IGA as parallel execution of MOGAs. To improve search efficiency of MOGAs, we added the following features.

In our new parallel execution architecture, in order to keep diversity of individuals among islands, we let one island use *normal selection* (*normal Island*) and remaining k islands use *biased selection* (*biased Island*) for multi-objective optimization problems with k objectives.

In general IGA, immigration operation exchanges small number of individuals in an island with its neighboring island. In our new architecture, immigrating individuals are chosen from k *biased Island* one by one every fixed period. Then they are compared and the individual which dominates others is selected. After that, we let the selected individual immigrate to *normal Island*. At the same time, one individual is selected in *normal Island*, and we duplicate it and let the duplicated individuals to immigrate to all *biased Islands*. The above immigration operation aims at preventing individuals from being selected in a particular island with the specific objective.

IV. Hardware Implementation of Proposed Method

In this section, we show two architectures suitable to hardware implementation of MOGAs based on our proposed

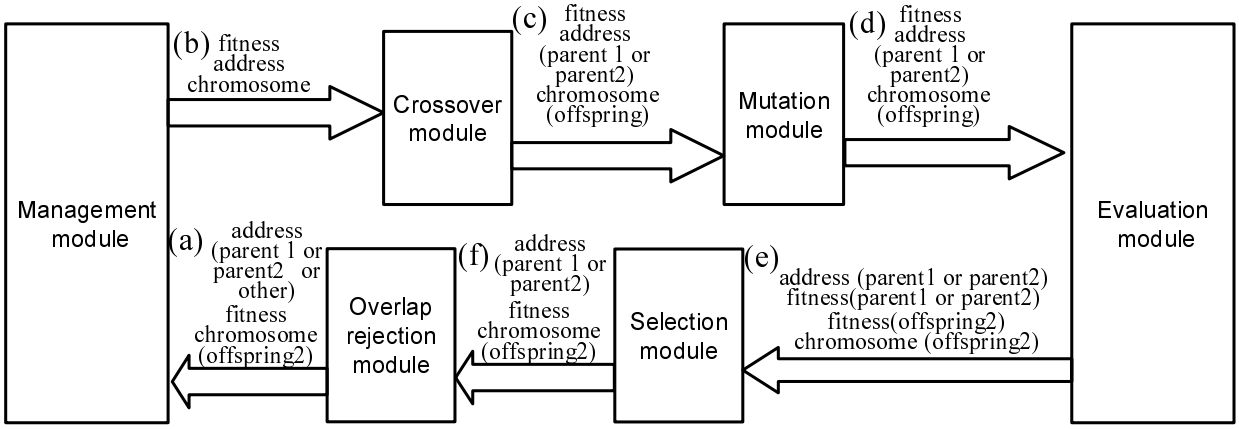

Fig. 1. Basic Architecture

method. One is called the *basic architecture* for constructing minimum GA circuits. The basic architecture composes a single pipeline consisting of several modules. These modules execute corresponding GA operations in a pipelined manner. We use the following modules: management module, crossover module, mutation module, evaluation module, selection module and overlap rejection module. The basic architecture is shown in Fig. 1.

The other one is called the *parallel architecture* for constructing parallel GA circuits. The parallel architecture composes multiple pipelines to be executed in parallel in an IGA manner. The parallel architecture consists of the following modules : management module, crossover module, mutation module, evaluation module, selection module, overlap rejection module, immigration module and relation module. The parallel architecture is shown in Fig. 2.

Management module retains population where each individual consists of fitness values and chromosomes. Crossover module, mutation module and evaluation module perform corresponding genetic operations. Selection module and overlap rejection module use techniques explained in Sect. III-B. Immigration module and relation module are used only in the parallel architecture.

In our proposed architectures, each module must be designed to receive and send out m bits of data (i.e.,chromosome) every clock (though it may require some extra clocks to output the first m bit data after the first m bit data is input). Buses between neighboring modules have width of m bits. Since each chromosome is coded as a fixed size string of n bits (n and m are given as parameters), $\lceil \frac{n}{m} \rceil$ clocks are required to process each chromosome, where $n \geq m$. Each module receives and processes data in a pipelined manner. We describe details of each module, hereafter.

A. Management Module

Management module includes memory in which the population is stored. This module reads individuals from the memory, and sends them to crossover module (Fig. 1 (b)), one by one. It also receives individuals from overlap rejection module (Fig. 1 (a)) and writes them to the memory.

When overlap rejection module requests management module to write an individual to the memory, chromosome and fitness values of the individual received from selection module are written to the specified address. Since memory read and write cannot be performed simultaneously, data of an individual cannot be sent to crossover module during memory write. Such a pipeline stall can be prevented by sending an offspring individual (which overlap rejection module retains) directly to crossover module. In the case when overlap rejection module does not request memory write, a randomly selected individual in the population, its address and fitness value are sent to crossover module (Fig. 1 (b)).

B. Crossover Module

Crossover module has a register r which retains chromosome, address and fitness value of the individual *parent1* received $\lceil \frac{n}{m} \rceil$ clocks before. Crossover module applies crossover operator to chromosome *parent1* and chromosome *parent2* which has just been received, and creates a new chromosome *offspring*. Crossover module also performs part of selection operation. The module compares fitness values of *parent1* and *parent2* and sends the address and fitness values of the superior individual. If two parents do not dominate each other, *parent1* is sent to crossover module.

C. Mutation Module

Mutation module applies mutation operator to the chromosome of *offspring* which is received from crossover module, and sends the chromosome of the resulting individual *offspring2* to the evaluation module. Also, the module sends the address and chromosome of *parent1* or *parent2* received from crossover module to evaluation module (Fig. 1 (d)).

D. Evaluation Module

Evaluation module calculates the fitness values of *offspring2* received from the mutation module using all objec-

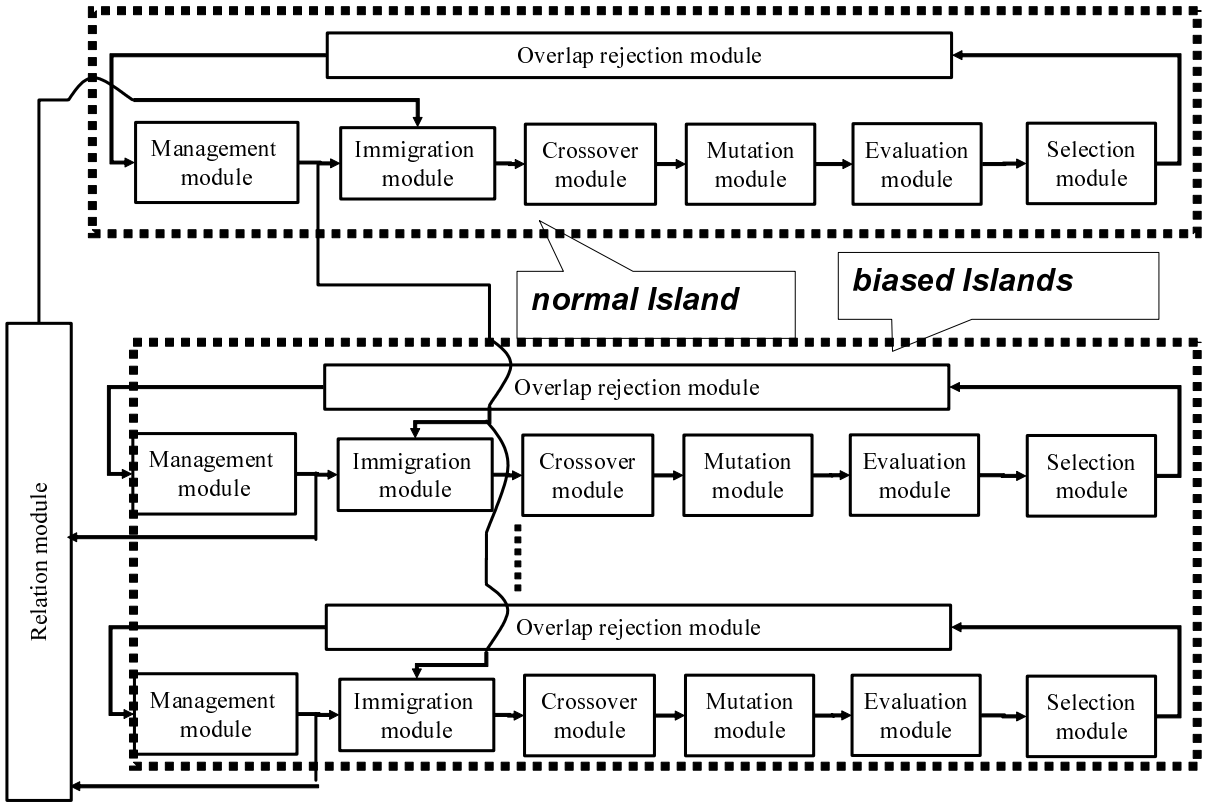

Fig. 2. Parallel Architecture

tive functions. Then, this module sends the calculated fitness values in addition to the information received from mutation module (Fig. 1 (e)).

E. Selection Module

Selection module uses one of two proposed selection methods described in Sect. III-B. The inputs of selection module are the chromosome and calculated fitness values of offspring, fitness values and address of the selected parent. Selection module compares fitness values of the selected parent and offspring, and then it decides if the offspring should be replaced with the parent or not. The result of decision is represented by a *selected flag*, which becomes true if the offspring should be replaced with the parent. Selection module transfers the fitness values and chromosome of offspring individual, *selected flag*, the address of the parent individual to overlap rejection module.

F. Overlap rejection module

Overlap rejection module updates population while removing redundant individuals. Removal of redundant individual is performed based on following three policies.

1) Offspring is replaced with parent in the population if the offspring is selected, except policy 2).
2) Offspring is not replaced with parent if population already has individual identical to the offspring. This is performed by comparing their fitness values, instead of their chromosomes.
3) If the offspring is not selected for replacement, and population includes identical individuals, one of these individual is replaced with the offspring.

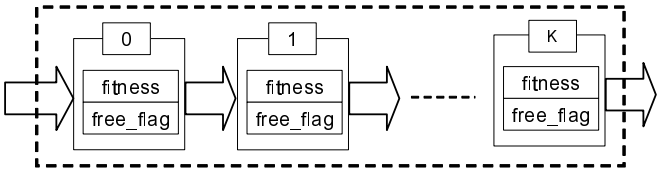

Fig. 3. Overlap Rejection Module

Overlap rejection module consists of k submodules. Variable k is the number of population size. Each submodule corresponds to each individual in the population (Fig. 3). Each submodule i has following parameters:

- Address of corresponding individual (i)
- Fitness values of corresponding individual ($fitness_i^p$)
- A flag ($free\_flag_i^p$)

$free\_flag_i^p$ is used to implement policy 3). This flag is set to true when more than two individuals with identical fitness values are found, and in this case, one of these individuals

will be replaced with offspring. The initial value of the flag is set to false.

Overlap rejection module receives parameter set of individual j from selection module. The parameter set consists of the fitness values $fitness_j^o$ and chromosome $chromosome_j^o$ of offspring individual, $selected\_flag_j^o$ and the address $overwrite\_adress_j^o$ of parent individual. Besides them, each offspring individual retains a flag This flag is set to true when an individual with fitness values identical to offspring individual is found in the population. The initial value of the flag is set to false.

The algorithm of submodule i is shown below. The input of the first submodule is given by selection module, and the output from the last module is disposed.

1) Submodule i receives parameters of individual j from submodule $i-1$ as follows: $fitness_j^o$, $chromosome_j^o$, $selected\_flag_j^o$, $overwrite\_adress_j^o$ and $found\_flag_j^o$.
2) If $found\_flag_j^o$ == false then goto 5.
3) If $fitness_i^p == fitness_j^o$ then $free\_flag_i^p \leftarrow$ true. (cancels replacement according to policy 2)
4) Goto 11.
5) If ($selected\_flag_j^o == true$ and $i == overwrite\_address_j^o$) == false then goto 7.
6) $fitness_i^p \leftarrow fitness_j^o$, $found\_flag_j^o \leftarrow$ true, goto 11. (this overwrites individual according to policy 1)
7) If $fitness_i^p == fitness_j^o$ then $found\_flag_j^o \leftarrow$ true, goto 11.
8) If ($free\_flag_i^p$ == true and $i! = overwrite\_address_j^o$) == false then goto 11
9) $fitness_i^p$ and $fitness_j^o$ are compared, and if individual i dominates individual j, then goto 11.
10) $fitness_i^p \leftarrow fitness_j^o$, $found\_flag_j^o \leftarrow$ true, $free\_flag_i^p \leftarrow$ false (this overwrites an individual according to policy 3).
11) Parameters of individual j are passed to the next submodule $i+1$.

G. Parallel Architecture

In the proposed parallelization method, one ordinary MOGA called *normal MOGA* and multiple *biased MOGA*s each of which takes one objective function prior to other objective functions are executed in the corresponding islands (pipelines). Basically we assign selection policies to islands so that the number of biased MOGAs is the same as the number of objective functions.

In order to exchange individuals between parallel MOGA islands, immigration module is introduced between management module and crossover module, as shown in Fig. 2. Most of time, we let crossover module receive individuals from management module from the same island, but we let it receive individuals from other island every specified interval.

Relation module chooses the best individual from individuals output from biased MOGA islands, and sends it to a normal MOGA island.

TABLE I
COMPARISON BETWEEN PROPOSED METHOD AND NSGA-II.

method	pareto solutions	evaluations per Island	total evaluations	processing time (sec)
normal method (1)	34.6	1000000	1000000	0.0100
normal method (2)	36.1	1000000	2000000	0.0100
normal method (4)	39.0	1000000	4000000	0.0100
normal method (6)	39.6	1000000	6000000	0.0100
biased method (3)	46.6	1000000	3000000	0.0100
NSGA-II	33.8	320000	320000	43.2

V. EXPERIMENTAL RESULTS

In order to evaluate search efficiency of our method, we have implemented the method as software and conducted some experiments with it. In the implementation, we used Half Uniform Crossover (HUX) and bit-mutation. As benchmark, the multi-objective Knapsack Problem [7] 2KP50-50 was used for evaluation. This problem has 52 pareto optimal solutions. Parameter values used in the experiment are as follows: the number of individuals per island is 64, crossover rate is 0.4, and mutation rate is 0.04. Experimental results are shown as average values of 10 trials.

The number of obtained pareto optimal solutions are shown in Fig. 4. We assume that MOGAs with our method are implemented as hardware circuits. So, we evaluated search efficiency as the number of obtained pareto optimal solutions per clock. In Fig. 4, "normal method" corresponds to the case when only the normal selection is used, and "biased method" does the case when both the biased selection and the normal selection are used together. The number in parentheses shows the number of islands (it shows the number of concurrent pipelines).

This result shows that our proposed method can generate part of pareto optimal solutions. We see that the cases with larger parallel degrees achieve higher performance and "biased method(3)" outperforms "normal method(6)". The obtained non-dominated solutions in the experiments are shown in Fig. 5, 6, 7, 8 and 9. Here, the number of evaluations per island is 1,000,000. These figures show the best results of 10 trials. In Fig. 5, "normal method(1)" mainly obtained non-dominated solutions around center part of pareto front. Normal methods with many island obtained wider variety of non-dominated solutions in pareto front, but they could not obtain pareto optimal solutions on both edges of pareto front. On the other hand, "biased method(3)" obtained the pareto optimal solutions on both edges. According to this discussion, we confirmed that executing biased selection together with normal selection concurrently is effective.

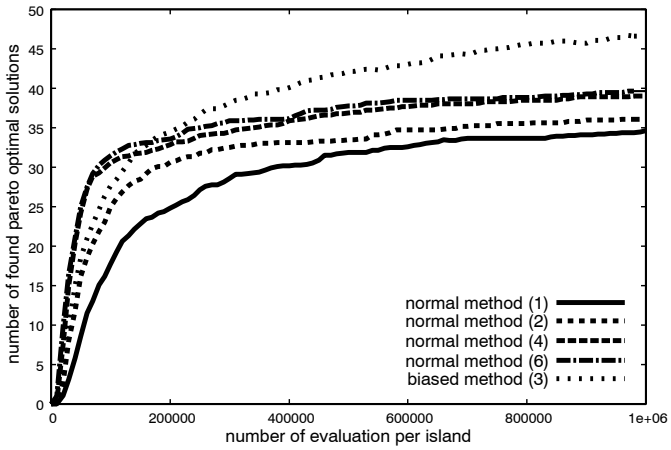

Fig. 4. Number of found pareto optimal solutions for Multi-Objective Knapsack Problem (2KP50-50).

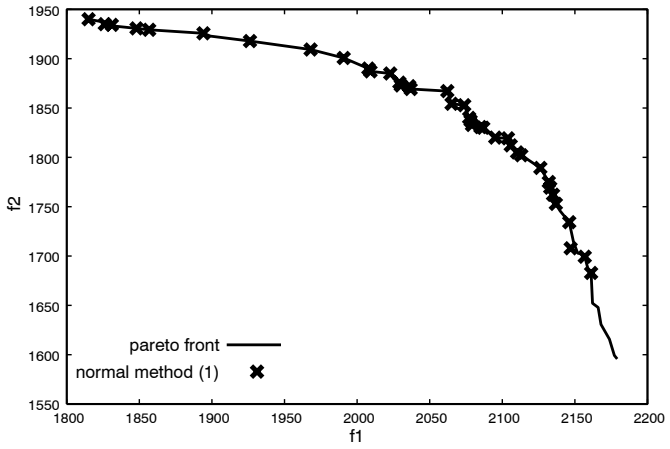

Fig. 5. Results of normal method (1) for Multi-objective Knapsack Problem (2KP50-50).

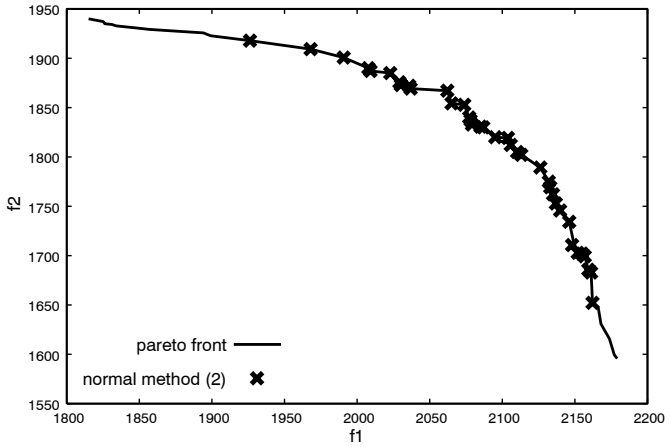

Fig. 6. Results of normal method (2) for Multi-objective Knapsack Problem (2KP50-50).

Next, we evaluated computation speed when our proposed method is implemented as hardware circuits. From our previous work [15], hardware circuits implemented on FPGA devices achieved 100–140MHz clock frequency. Since hardware architecture proposed in this paper is similar to our previous method, we assumed that our circuits can achieve at least 100MHz clock frequency. In our proposed method, evaluation of each individual can be executed in one clock cycle per island. So, we can estimate that one evaluation consumes 1.00×10^{-8} seconds. With this assumption, we compared our proposed method with NSGA-II. Here, NSGA-II is implemented as software. Experimental environments are as follows: We used gcc version 3.35 with optimized option O3 and executed the software on a general PC with Pentium 4 (2.4GHz), 256MB memory on linux 2.6.10. We used the following parameter values: the number of individuals is 64; crossover rate is 0.6; and mutation rate is 0.02. AS a result, software implementation of NSGA-II consumed 1.53×10^{-4} seconds for each evaluation. So, we see that our proposed method achieves much higher performance than NSGA-II.

Table I shows the number of obtained pareto optimal solutions by both methods. Similarly, the number of evaluations and processing time are shown in this table. Here, the numbers of evaluations are the values when GAs converge. Processing time of our proposed method is calculated based on the assumption explained above.

From this result, we confirmed that our proposed methods (normal methods and biased method) outperforms NSGA-II, and especially our biased method achieves good performance.

VI. Conclusions

In this paper, we proposed a method to implement MOGAs as efficient hardware circuits. Novelty of our method is in our selection mechanism to keep diversity of individuals by efficiently checking superiority among individuals considering multi objectives, and our architecture which is scalable to increase the number of concurrent pipelines. In experiments, we compared search efficiency of MOGAs implemented by our proposed method with existing method NSGA-II, and confirmed that our MOGAs work much more efficiently.

As part of future work, we design *interface module* to exchange input/output data between our MOGA circuits and other circuits such as CPU. Then we are planning to implement MOGAs on FPGA using the proposed method, to apply our method to various multi objective optimization problems, and to improve execution efficiency of the resulting circuits.

References

[1] Barry Shackleford, Etsuko Okushi, Mitsuhiro Yasuda, Hisao Koizumi, Katsuhiko Seo, Takahashi Iwamoto and Hiroto Yasuura, High-performance hardware design and implementation of genetic algorithms, Hardware implementation of intelligent systems, pp.53–87, 2001.

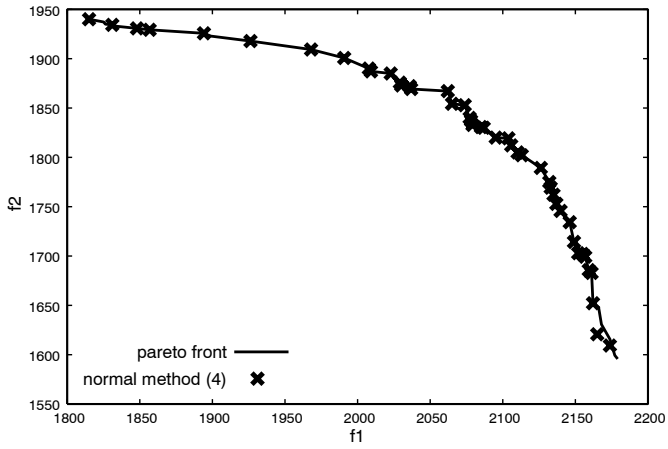

Fig. 7. Results of normal method (4) for Multi-objective Knapsack Problem (2KP50-50).

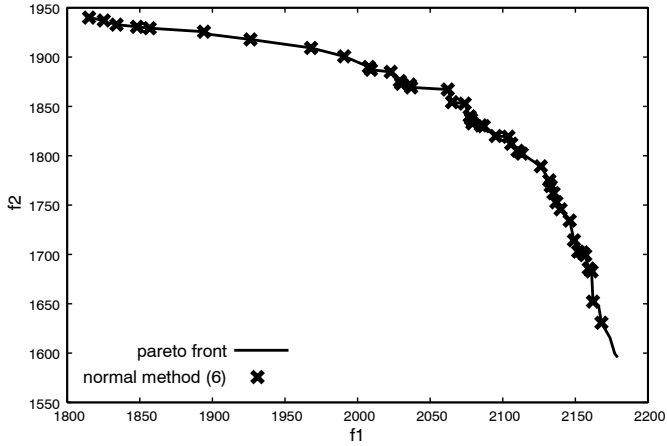

Fig. 8. Results of normal method (6) for Multi-objective Knapsack Problem (2KP50-50).

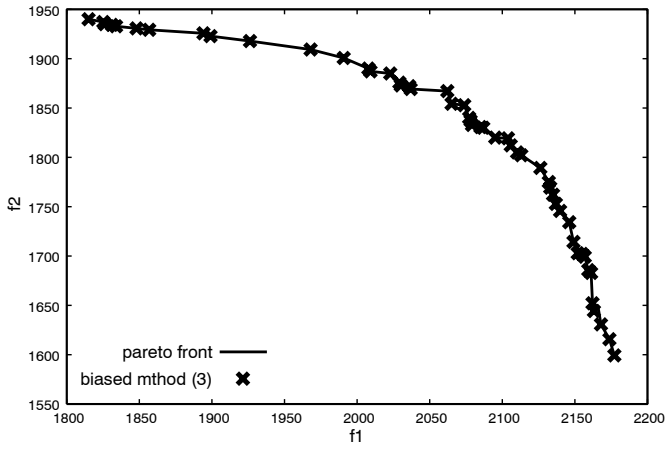

Fig. 9. Results of biased method (3) for Multi-objective Knapsack Problem (2KP50-50).

[2] Chatchawit Aporntewan and Prabhas Chongstitvatana, A Hardware Implementation of the Compact Genetic Algorithm, In Proc. of the 2001 Congress on Evolutionary Computation (CEC2001), pp.624–629, 2001.

[3] Deb, K., S. Agrawal, A. Pratap, and T. Meyarivan , A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II, In Proc. of the Parallel Problem Solving from Nature VI, pp.849–858, 2000.

[4] Erick Cantú-Paz, A Survey of Parallel Genetic Algorithms, Technical Report 97003, Illinois Genetic Algorithms Laboratory, 1997.

[5] Hiroshi Satoh, Isao Ono and Shigenobu Kobayashi, Minimal Generation Gap Model for GAs Considering Both Exploration and Exploitation, In Proc. 4th Int'l Conf. on Soft Computing (IIZUKA'96), pp.494–497, 1996.

[6] Hisashi Shimodaira, An Empirical Performance Comparison of Niching Methods for Genetic Algorithms, IEICE Trans. Inf. & Syst., vol.E85-D, no.11, pp.1872–1880, 2002.

[7] MCDM Numerical Instances Library, `http://www.univ-valenciennes.fr/ROAD/MCDM/ListMOKP.html`

[8] Michael Guntsch, Bernd Scheuermann, Hartmut Schmeck, Martin Middendorf, Oliver Diessel, Hossam ElGindy and Keith So, Population based Ant Colony Optimization on FPGA, Proc. of 2002 IEEE Int'l. Conf. on Field-Programmable Technology (FPT2002), pp.125–133, 2002.

[9] Michael Wrighton and Andre DeHon, Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement, In Proc. of 2003 ACM Int'l. Symp. on Field Programmable Gate Arrays (FPGA2003), pp.33–42, 2003.

[10] Osamu Kitaura, Hideaki Asada, Motoaki Matsuzaki, Takamitsu Kawai, Hideki Ando and Toshi Shimada, A Custom Computing Machine for Genetic Algorithms without Pipeline Stalls, In Proc. of 1999 IEEE Int'l Conf. on Systems, Man, and Cybernetics (SMC'99), vol. V, pp.577–584, 1999.

[11] Ryoichi Kobayashi, Masahide Abe,and Masayuki Kawata, A Hardware Implementation of Genetic Algorithm for Extraction of Disconnected Closed Loops Using FPGAs, *IEICE Tech. Rep.*, CS2000-151, pp.29–36, 2001 (in Japanese).

[12] Sadiq M. Sait and Habib Youssef, Iterative Computer Algorithms with Applications in Engineering, pp.109–181, THE IEEE COMPUTER SOCIETY, 1999.

[13] Shin'ichi Wakabayashi, Tetsushi Koide, Koichi Hatta, Yoshikatsu Nakayama, mutsuaki Goto, Naoyoshi Toshine, An LSI Implementation of a Genetic Algorithm with Adaptive selection of Crossover Operators, IPSJ Journal, vol.41, no.6, pp.1766–1776l, 2000 (in Japanese).

[14] Shin'ichi Wakabayashi, Tetsushi Koide, Naoyoshi Toshine, Masataka Yamane, Hajime Ueno, Genetic algorithm accelerator GAA-II, In Proc. 2000 Asia-South Pacific Design Automation Conf. (ASP-DAC2000), University LSI Design Contest, pp.9–10, 2000.

[15] Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata, Keiichi Yasumoto, Minoru Ito, General Architecture for Hardware Implementation of Genetic Algorithm, In Proc, of the Fourteenth Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM2006), 2006 (to appear).

[16] Tomoya Kitani, Yoshifumi Takamoto, Keiichi Yasumoto, Akio Nakata and Teruo Higashino, A Flexible and High-Reliable HW/SW Co-Design Method for Real-Time Embedded Systems, In Proc. of 25th IEEE Int'l. Real-Time Systems Symp. (RTSS 2004), pp.437-446, 2004.

[17] Zitzler, E., Laumanns, M., and Thiele, L. , SPEA2: Improving the Strength Pareto Evolutionary Algorithm, Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 2001.