General Architecture for Hardware Implementation of Genetic Algorithm

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

[†] Shiga University
Hikone, Shiga 522-8522, Japan
shibata@biwako.shiga-u.ac.jp

1 Introduction

In this paper, we propose a technique to flexibly implement Genetic Algorithms (GAs) for various problems on FPGAs. For the purpose, we propose a common architecture for GA. The proposed architecture allows designers to easily implement a GA as a hardware circuit consisting of parallel pipelines which execute GA operations. The proposed architecture is scalable to increase the number of parallel pipelines. The architecture is applicable to various problems and allows designers to estimate the size of resulting circuits. We give a model for predicting the size of resulting circuits from given parameters. Based on the proposed method, we have implemented a tool to facilitate GA circuit design and development. Through experiments using Knapsack Problem and Traveling Salesman Problem (TSP), we show that the FPGA circuits synthesized based on the proposed method run much faster and consume much lower power than software implementation on a PC, and that our model can predict the size of the resulting circuit accurately enough.

2 Genetic Algorithms

GA is a technique for efficiently finding near optimal solutions for combinatorial optimization problems. GA uses multiple *individuals* (i.e., candidate solutions) where each individual includes a chromosome representing a point in a search space of a given problem. GA works as follows: (1) Individuals are generated with randomly decided chromosomes. The set of individuals is called *population*; (2) The *fitness value* is calculated for each individual. The fitness value represents how close to the optimal solution the individual is; (3) The *selection* operation selects a certain number of individuals with better fitness values from population. (4) The *crossover* operation is applied to generate new individuals. (5) The *mutation* operation is applied to mutate the chromosome of the new individuals at a certain probability. The above operations from (2) to (5) are repeatedly applied specified times or until a good approximation close to the optimal solution is obtained.
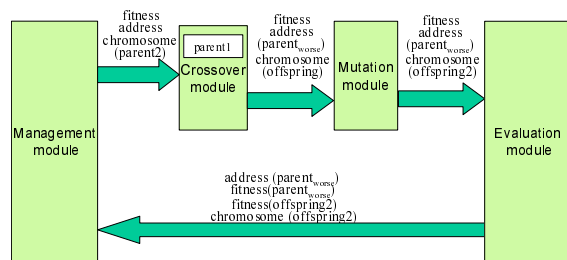

Figure 1. Basic Architecture

3 Proposed Architecture

The proposed architecture is composed of *Basic Architecture* corresponding to a single pipeline of GA operations and *Parallel Architecture* which combines multiple GA pipelines efficiently.

In the basic architecture, processes of GA are divided into four submodules named management module, crossover module, mutation module and evaluation module, as shown in Figure 1. Each chromosome is coded as a string of n bits. Buses between neighboring modules have width of m bits. n and m are given as parameters.

Each module is designed so that it receives m bits of data every clock, and outputs m bits of data every clock (it may take some clocks to output the first m bit data after the first m bit data is input). Therefore, $\lceil \frac{n}{m} \rceil$ clocks are used to process each chromosome, where $n \geq m$. Each module receives and processes data in pipelined manner.

We adopt simplified Minimal Generation Gap (MGG) model [1] to reduce required memory amount. In this model, two individuals are picked up from the current population. Crossover and mutation operations are applied to these individuals to generate a new offspring. This offspring individual is then evaluated. Selection operation selects the individuals with the higher fitness values from the family (an offspring individual and the parent individuals) and removes the worst individual in the family.

The management module stores the population in memory. As shown in Figure 1, following items are received from the evaluation module: the address and the fitness

value of the parent individual with lower fitness value ($parent_{worse}$), and the chromosome of the newly generated individual and its fitness value. The fitness value of $parent_{worse}$ is compared with that of the new individual. If fitness value of the new individual is higher than that of the $parent_{worse}$, chromosome and fitness value of the new individual are overwritten to $parent_{worse}$ in the memory and they are sent to crossover module. Otherwise, chromosome, fitness value and address of randomly selected individual are sent to crossover module.

The crossover module has a register r which retains the chromosome, the address and the fitness value of the latest individual received from the management module. It applies the crossover operator to the chromosome received from the management module (*parent2*) and the chromosome retained in r (*parent1*), and generates a new chromosome *offspring*. The crossover module compares fitness values of *parent1* and *parent2*, and sends the address and the fitness value of $parent_{worse}$ to the mutation module. The chromosome of *offspring* is also sent to the mutation module.

The mutation module applies the mutation operator to the chromosome of *offspring* and sends the chromosome of the resulting individual (*offspring2*) to the evaluation module. Also, the module sends the address and the fitness value of $parent_{worse}$ to the evaluation module.

The evaluation module calculates the fitness value of the new individual *offspring2*. Also, this module sends following items to the management module: the address and the fitness value of $parent_{worse}$ and the chromosome and the fitness value of *offspring2*.

In the parallel architecture, the GA pipeline developed based on the basic architecture is regarded as an island of the IGA model [2], and the individual exchange mechanism between neighboring GA pipelines is implemented. The *immigration module* is inserted between the management module and the crossover module. The immigration module is connected to the management modules of its GA pipeline and the neighbor pipeline, and it periodically receives individuals from the neighbor GA pipeline.

4   Prediction Model

In this section we describe the prediction model of GA circuit size. According to preliminary experiment, we confirmed that we can predict the size of each module with a linear function of a problem size and a population size. Because we adopt simple communication interface between modules, we see that approximate total circuit size can be calculated as the sum of the sizes of modules used in the whole circuit. In our approach, we first synthesize modules with different problem sizes, and then obtain linear functions using multiple regression.
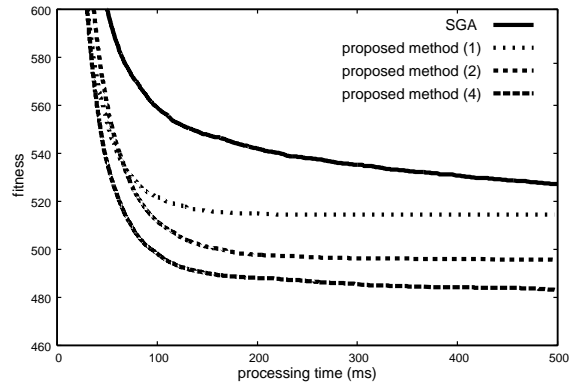

Figure 2. search efficiency (TSP, eil51)

5   Experimental Results and Evaluation

In this section, we show the performance of the circuits implemented based on our proposed architecture and the accuracy of our prediction model through experiments.

We conducted logic synthesis for the circuit descriptions using Altera Quartus II. We used Altera Cyclone FPGA devices as target devices. We used 64bit Knapsack Problem and a TSP instance called eil51. We compared the circuits implemented based on our method with software implementation of traditional GA (see Sect. 2) executed on a PC with Pentium4 2.4GHz and 256MB memory. Hereafter we refer to the software implementation as SGA.

At first, we measured how good solutions can be obtained within specified execution time for our circuits and the SGA. For Knapsack Problem, we confirmed that our circuits achieve much better performance than SGA. The results of TSP are shown in Figure 2. In Figure 2, the lower fitness value means the better solution. The number in parentheses indicates the number of concurrent pipelines in the circuit. Figure 2 shows that with our circuits candidate solutions converge to semi-optimal solutions much more quickly than SGA and that the quality of solutions in our circuits is much better than SGA as long as the same crossover is used. It also shows that the performance of our circuits can be greatly improved by increasing the number of GA pipelines. We also confirmed that the generated circuit consumes 1/80 of TDP (Thermal Design Power) of Pentium4 2.4GHz, at most. Finally, we evaluated our prediction model and confirmed that prediction error is within 3% at the maximum for both Knapsack Problem and TSP.

References

[1] Hiroshi Satoh, Isao Ono and Shigenobu Kobayashi, Minimal Generation Gap Model for GAs Considering Both Exploration and Exploitation, Proc. IIZUKA'96, pp.494 –497, 1996.

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