Flexible Implementation of Genetic Algorithms on FPGAs

Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata†, Keiichi Yasumoto and Minoru Ito
Grad. Sch. of Info. Sci., Nara Inst. Sci. and Tech.
Ikoma, Nara 630-0192, Japan
{tatsu-ta,yosihi-m,yasumoto,ito}@is.naist.jp

† Dept. of Info. Proc. and Man., Shiga Univ.
Hikone, Shiga 522-8522, Japan
shibata@biwako.shiga-u.ac.jp

ABSTRACT

In this paper, we propose a technique to flexibly implement genetic algorithms for various problems on FPGAs. For the purpose, we propose a basic architecture for GA which consists of several modules for GA operations to compose a GA pipeline, and a parallel architecture consisting of multiple concurrent pipelines. The proposed architectures are simple enough to be implemented on FPGAs, applicable to various problems such as Knapsack Problem and Traveling Salesman Problem (TSP), and easy to estimate the size of the resulting circuit. We also propose a model for predicting the size of resulting circuit from given parameters consisting of the problem size, the number of concurrent pipelines, and the number of candidate solutions for GA. Based on the proposed method, we have implemented a tool to facilitate GA circuit design and development. This tool allows designers to find appropriate parameter values so that the resulting circuit can be accommodated in the target FPGA device, and to automatically obtain RT-level VHDL description. Through experiments using Knapsack Problem and 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, that the achievable performance can be improved as the size of the target FPGA device increases, and that our model can predict the size of the resulting circuit accurately enough.

Categories and Subject Descriptors

B6.3 [**Logic Design**]: Design Aids - Automatic synthesis

General Terms

Design, Experimentation

Keywords

genetic algorithm, FPGA, hardware design automation, Knapsack Problem, Traveling Salesman Problem

1. INTRODUCTION

Genetic Algorithm (GA) is a technique for efficiently finding near optimal solutions for combinatorial optimization problems. Since GAs are easy to implement and work efficiently enough, they have been used for various practical problems such as scheduling, design and allocation. On the other hand, it is known that GAs require larger computation power than algorithms designed exclusively for particular problems. Applications of GA can be categorized into several types : the first type includes the optimal route decision problem such as TSP and analogous problems for obtaining optimal routes satisfying time restrictions which can be used in personal navigation system for sightseeing tours [1] or parcel delivery. Examples of the second type applications include calculating multicast delivery tree in large-scale network which optimizes more than one QoS metrics (e.g., delay and cost) [2]. The third type includes high quality video compression algorithm [3] which has been standardized as ISO JBIG2. The first type of applications may be used through portable computing devices such as cellular phones and PDAs. The second and third types of applications are typically executed on routers and information appliances (e.g., HDTV, facsimile, etc), respectively. These devices usually have relatively low-cost micro processor and implementing GA applications as software and executing on these devices is not realistic in terms of cost and power consumption. Hardware implementation of GA on FPGA could be more realistic in these cases.

There are several research efforts for hardware implementation of GAs [4, 5, 6, 7]. However, these existing techniques use specific architectures for the target problems, e.g., for TSP [4] or for rather simple problems such as Set Coverage Problem [5]. In order to make GAs on information appliances and small electronic devices for various purposes, we need a general architecture for hardware implementation of GAs. We also need a method to optimize the resulting circuit for a given problem and a target device, satisfying various constraints such as required performance, cost and power consumption.

In this paper, we propose a flexible implementation technique of GA on FPGAs with cost-performance tradeoffs. In the proposed method, we first propose a general basic architecture for GA consisting of several modules which cooperatively execute GA operations in a pipeline. We call this sequential composition of the modules a GA pipeline. In our basic architecture, simplified GA operations and generation model are used so that various GA problems can be developed with this architecture. The speed and size (including memory) of the resulting circuit can be also optimized by this architecture. Then we propose a parallel architecture which concurrently executes GA pipelines, exchanging candidate solutions among GA pipelines.

In our synthesis method, the problem size (number of bits in each candidate solution), the number of candidate solutions, and the number of parallel pipelines can be specified as parameters. In order to optimize the performance of the resulting circuit for the target FPGA device, we propose a model for predicting the size of the resulting circuit from the given parameter values. As we show through experiments, the prediction result is accurate enough for practical use. This accuracy of prediction is achieved by constructing whole circuit as a simple combination of multiple GA pipelines,

.

and each pipeline as a simple combination of several modules and control circuits.

We have implemented a tool to facilitate GA circuit design and development. This tool consists of two parts: circuit size check part and circuit derivation part. The circuit size check part utilizes a prediction model and calculates parameter values with which the hardware circuits can be synthesized on a specified FPGA device. It can also check whether the size of the circuit with the given parameter values is within the target FPGA device. The circuit derivation part generates the RT level VHDL description when the parameter values are given. This tool would be helpful for designers especially in early phase of design since various combinations of parameter values can be tested to check if those values can generate the circuits which fully utilize the target FPGA devices.

In order to show applicability of the proposed method, we give detailed design of two example GAs for Knapsack Problem and TSP. Through experiments, we show that the FPGA circuits synthesized based on the proposed method run much faster and consume much lower power than software implementation executed on Pentium 4, that the achievable performance can be improved as the size of the target FPGA device increases, and that our model can predict the size of the resulting circuit accurately for practical use.

In the following Sect. 2, studies on hardware GA is briefly presented. In Sect. 3, we address the outline of the proposed method. Sect. 4, describes the proposed architectures for hardware implementation of GA. In Sect. 5, we design two example GAs, Knapsack Problem and TSP with our architecture. Sect. 6 presents a model for predicting the size of the resulting circuit from given parameters. Sect. 7 and Sect. 8 describe our design support tool and the experimental results, respectively. Finally, we conclude the paper in Sect. 9.

2. RELATED WORKS

Several hardware implementation techniques for GAs have been proposed so far. In [5], Barry et al. produced hardware circuits for Set Coverage Problem using a technique called the steady-state GA. In their report, the circuits run 2200 times faster than a software implementation on a workstation with 100MHz CPU. Aporntewan et al. proposed a hardware implementation technique for Compact Genetic Algorithm on FPGAs[6]. This algorithm generates new candidate solutions from the probability distribution of former candidate solutions. This approach reduces required memory, but it can only be applied to simple problems such as one-max problem. Wakabayashi et al. synthesized a hardware circuit called GAA-II [7] which works very well for a benchmark problem called Dejong's test function. In [4], Graham et al. implemented a GA for TSP on a board consisting of four Xilinx 4010 FPGAs and external memory. They reported that their circuit works 11 times faster than software implementation on a workstation with 125MHz PA-RISC.

These existing methods aim at achieving higher performance than software implementation, and most of them use architectures exclusive for the target problems. Thus, implementing hardware GAs for other problems based on these methods is difficult.

In order to implement hardware GAs for various problems on FPGAs so that it can be used in information appliances, small electronic devices, and so on, a general architecture suitable for hardware implementation of GAs is necessary. Also, the synthesized circuit have to satisfy conditions regarding to performance, cost, and power consumption for the target device. For flexible implementation of hardware circuits mentioned above, Kitani et al. proposed an efficient design method for real-time embedded systems [8]. This method allows designers to select an appropriate set of hardware components which satisfies conditions regarding to performance and the total cost of the system. However, it is difficult to apply this method for hardware GA since it does not treat the case where the component size changes depending on the problem size.

3. BASIC IDEAS FOR HARDWARE IMPLEMENTATION OF GA

In this section, we give a brief explanation about GA and then show our basic ideas for implementing GA as a hardware circuit.

3.1 Genetic Algorithms

GA uses multiple *individuals* (i.e., candidate solutions) where each individual includes a chromosome representing a point in a search space of the 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 is applied to the population and a certain number of individuals with better fitness values are selected; (4) The *crossover* operation is applied to pairs of the selected individuals to generate new individuals where the chromosome of a new individual is generated by mixing parents' chromosomes; (5) The *mutation* operation is applied to the new individuals. These new individuals are replaced with the individuals which are not selected in (3). The above operations from (2) to (5) are repeatedly applied specified times or until a good approximation close to the optimal solution is obtained.

An individual which does not represent any point in a search space is called a *lethal* individual. Here, we give an example of lethal individual in Knapsack Problem. In Knapsack Problem, multiple items and one knapsack are given as inputs. Different values and volumes are assigned to items, and the knapsack has fixed capacity. The objective is to find the most valuable set of items that can be accommodated in the knapsack. If the sum of volumes of items in a candidate solution expressed by a chromosome exceeds the capacity of the knapsack, the individual which contains the chromosome is a lethal individual.

3.2 Basic Ideas to Synthesize Hardware Circuits for GAs

The goal of the proposed method is to synthesize an efficient hardware circuit of a GA for a given problem and problem size (size of each solution) which fully utilize the target FPGA device. The numbers of logic elements and memory blocks are taken into account when deciding if the circuit is accommodated in the FPGA device. To achieve this goal, we need (1) an efficient and general architecture for hardware implementation of GAs, (2) a technique for improving performance of the resulting circuit by parallel execution of GA, and (3) a technique to predict the size of the resulting circuit. For the above (1), efficient memory utilization is essential. So, we adopt a special generation model based on the MGG model [9]. The outline of this generation model is explained in Sect. 3.3. For the above (2), the synchronization among multiple parallel pipelines should not be too frequent. We adopt the island GA model [10] as the parallel execution architecture. The brief outline of the island GA model is explained in Sect. 3.4. The details of our basic architecture and parallel architecture are explained in Sect. 4. For the above (3), we compose the architecture as simple as possible and develop a model to predict the size of the resulting circuit from the problem type, the problem size, the number of candidate solutions, and the number of parallel pipelines. The detail is explained in Sect. 6.

3.3 Generation Model Used in Our Method

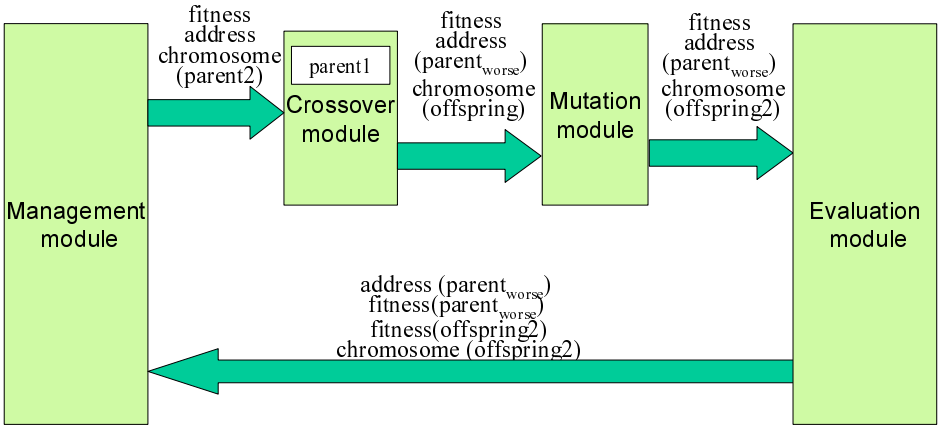


Figure 1: Basic Architecture

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 [5] and [6], 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 our method, we use Minimal Generation Gap (MGG) generation model [9]. In MGG model, two individuals are picked up from the current population. Crossover and mutation operations are applied to these individuals to generate new individuals. These new individuals are then evaluated, and individuals with good fitness values are selected using roulette selection[11] from the family (new individuals and the parent individuals). These selected individuals replace original individuals for the next generation population. Roulette selection is a technique to select individuals with probabilities decided based on their fitness values. In our method, however, instead of using roulette selection, we adopt a simpler selection mechanism which always selects the individual with the highest fitness value and replaces the worst individual in the family with it. This simplification makes it easy to construct pipelined and parallel circuit for processing individuals as well as greatly reduces the required memory for storing individuals.

It is possible that utilization of MGG model results in decreased performance. In Sect. 8, we investigate the performance of the circuit based on our method and software implementation of the ordinary generation model.

3.4 Outline of Island GA

There are various techniques for parallel execution of GA. In this paper, we use the technique of island GA (IGA, hereafter). 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 fall 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, what each island (processing unit) should do for parallel execution is only exchanging individuals with its neighboring island. So, the synchronization mechanism becomes very simple and does not depend on the number of parallel processing units. This greatly contributes the scalability of the resulting circuits.

4. GENERAL ARCHITECTURES FOR HARDWARE GA

In this section, we describe the *basic architecture* and the *parallel architecture* as general architectures to implement hardware GA. Our basic architecture contains a GA pipeline which processes one generation of operations with several modules in a pipeline, based on the special generation model explained in Sect. 3.3. In our parallel architecture, multiple GA pipelines developed in the basic architecture are executed in parallel based on island GA model in Sect. 3.4.

4.1 Basic Architecture

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 Fig. 1. Each chromosome is coded as a string of n bits. Buses between each 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. Hereafter, we describe details of each module.

Management module

The management module includes memory, and stores the population in it. The module also reads individuals from memory and sends them to the crossover module (step1), and receives individ-

uals from the evaluation module and write them to the memory (step2). As shown in Fig. 1, in step1, following items are sent to the crossover module : a randomly selected individual, its address and fitness value. Also, following items are received from the evaluation module : the address and the evaluation value of the parent individual with lower evaluation value, the chromosome of the newly generated individual and its evaluation value. In step2, the fitness value of the received parent individual is compared to that of the new individual. Chromosome and fitness value of the new individual is overwritten to the parent individual only if fitness value of the new individual is higher than that of the parent individual.

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 $offspring$. In the proposed method, only one new chromosome is generated when the crossover operator is applied, in order to reduce memory space.

To follow the generation model described in Sect. 3.3, $offspring$ has to be overwritten to the parent with lower fitness value (denoted by $parent_{worse}$), if $offspring$ has higher fitness value than $parent_{worse}$. In order to do this operation efficiently, 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 (Fig. 1).

Mutation module

The mutation module applies the mutation operator to the chromosome of $offspring$ which is received from the crossover operator, and sends the chromosome of the resulting individual (denoted by $offspring2$) to the evaluation module. Also, the module sends the address and the chromosome of $parent_{worse}$ received from the crossover module to the evaluation module.

Evaluation module

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

4.2 Parallel Architecture

In the parallel architecture, the GA pipeline developed based on the basic architecture is regarded as an island of the IGA model, and the individual exchange mechanism between neighboring GA pipelines is implemented.

For this purpose, the *immigration module* is inserted between the management module and the crossover module, as shown in Fig. 2. 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. Since the immigration module is connected to two management modules independently of the number of parallel GA pipelines, the number of pipelines does not affect the length of the critical path.

4.3 Coping with varying processing latency of a module

For some problems like TSP or Job Shop Scheduling Problem, some modules (e.g., the crossover module) in a GA pipeline are required to process complex data which may take variable number of clocks. This reduces the performance of the GA pipeline due to pipeline stall. This situation can be avoided by the following two techniques.

The first technique is to duplicate a module with varying latency and make the duplicated modules run concurrently so that data can be sent to the idle module, avoiding busy modules. By preparing enough number of extra modules for a variable latency module, the duplicated modules can achieve the constant processing time as a total.

The second technique is to insert a buffer between a variable latency module and its next module in a GA pipeline. When the variable latency module finishes data processing, the module writes output data on the buffer and start processing for the next data. The next module receives data from this buffer when it needs data. By preparing enough size of a buffer, variable latency of a module can be averaged.

5. PROBLEM-SPECIFIC DESIGN

In this section, we design modules used in our basic architecture for Knapsack Problem and TSP.

5.1 Knapsack Problem

For Knapsack Problem with s items, each chromosome can be represented by a s bit binary string, where i-th gene of the chromosome represents whether i-th item is in the knapsack or not. For the sake of simplicity, we suppose that s bits of each chromosome are transferred between modules every clock. We can easily modify the modules so that m bits ($m \leq s$) of a chromosome are transferred every clock cycle. In that case, each module takes $\lceil \frac{s}{m} \rceil$ clock cycles to transfer or receive a chromosome.

Management module

We let the management module retain multiple s bit chromosomes in the memory. A chromosome is selected at random and transferred to the immigration module (to the crossover module if the number of pipelines is one).

Crossover module

We use the uniform crossover technique, that is, for each i ($1 \leq i \leq s$), i-th gene of the child individual is copied from the i-th gene of either of two parent individuals. The decision is made at random.

Mutation module

For each i ($1 \leq i \leq s$), the value of the i-th gene is reversed with the probability given as a mutation rate.

Evaluation Module

The fitness value is calculated as follows.

- The fitness value is 0 if the total sum of volumes of all items in the knapsack exceeds its capacity.
- Otherwise, the fitness value is the total sum of values of all items in the knapsack.

The evaluation module retains the value and the volume of i-th item in registers $reg\_value[i]$ and $reg\_volume[i]$, respectively. We assume that the values and volumes of all items are given and stored in these registers in advance.

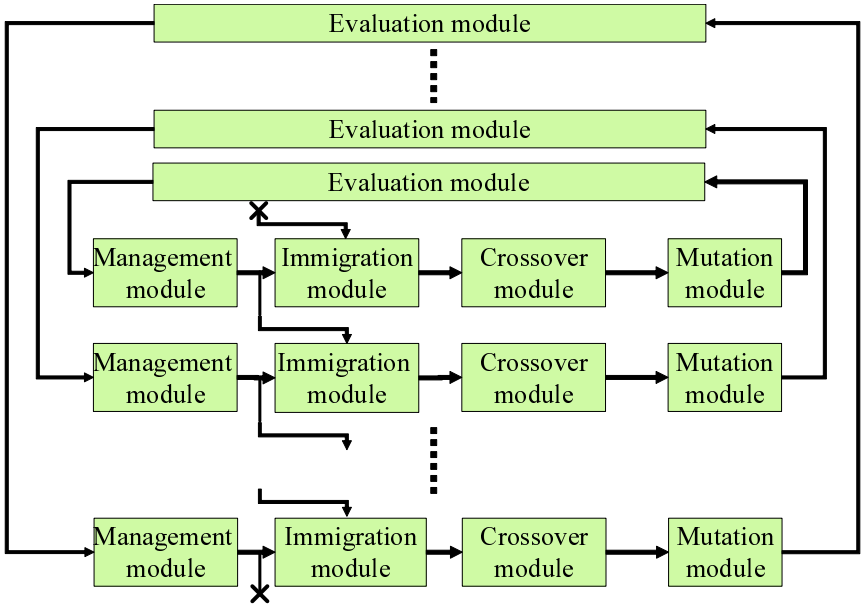


Figure 2: Parallel Architecture

In order to shorten the critical path, the total sums of values and volumes of all items in the knapsack are calculated in a bottom-up manner based on a binary tree where two values (volumes) are added for each of $\lfloor \frac{s}{2} \rfloor$ pairs at the leaves of the tree, for each of $\lfloor \frac{s}{4} \rfloor$ pairs of their sums at the next level of the tree, and so on. Thus, the calculation requires $\log_2 s$ clock cycles. However, since adders and registers in j-th depth ($q \leq j \leq log\ s$) in the tree are not used simultaneously with those in k-th depth ($k \neq j$), the calculation can be pipelined so that one fitness value is output every clock cycle.

After calculation of the sums of values and volumes, the fitness value is calculated and transferred to the management module.

Parallel Execution of GA Pipelines

When we want to execute multiple GA pipelines in parallel, we use the immigration module in each pipeline so that chromosomes are exchanged between pipelines. The above modules designed for Knapsack Problem output a chromosome every clock cycle. In order to exchange chromosomes between pipelines, we let each management module output a chromosome to two immigration modules (one is in the same pipeline and another is in the next pipeline) at the same time. We let each immigration module usually receive a chromosome from the management module in the same pipeline, but receive from the previous pipeline at a given period. For the purpose, we use a counter which is decremented every clock cycle and let the immigration module to receive from another pipeline only when the counter becomes 0.

5.2 TSP: Traveling Salesman Problem

In TSP, each chromosome is coded as a binary sequence with $n \times m$ bits, where n is the number of cities and m is the number of bits to represent each city. For example, when we treat TSP with 51 cities, we use 6 bits to represent each city and 306 bits to represent the whole chromosome. The fitness value is represented by a fixed number of bits, for example, by a 16 bit integer.

Management Module

In general, the size of a chromosome in TSP is likely to become large. So, we let modules transfer and receive a chromosome in multiple clock cycles. Hereafter, we suppose that each chromosome is transferred by m bits in n clock cycles.

Crossover Module

We implement a crossover module based on PMX [11] since it never generates lethal individuals and it is simple enough for hardware implementation.

First, PMX decides a sub-region in a chromosome between two random loci (i.e., positions in a chromosome). A new chromosome is generated by concatenating genes in a sub-region of one parent and genes from the other parent. This is called two point crossover. In a new (child) individual, the same city might appear more than once in its chromosome. Such duplicated cities are replaced with cities not included in the chromosome, so that each city appears in the chromosome only once.

The crossover module includes three sets of registers named $PA1$, $PA2$ and RS whose sizes are $n \times \lceil \log_2(n) \rceil$ bits. Each register set includes n registers with $\lceil \log_2 n \rceil$ bits. These registers are denoted as $PA1[i]$, $PA2[i]$ and $RS[i]$, respectively, where $i \in \{0, ..., n-1\}$. $PA1$ and $PA2$ retain chromosomes received from the management module. Genes at k-th locus of parent individuals are stored in $PA1[k]$ and $PA2[k]$.

The crossover operation starts after two chromosomes of parent individuals are stored in $PA1$ and $PA2$ completely. First the crossover module generates two random numbers $N1$ and $N2$ such that $N1 \leq N2 \leq n$. Then, the following steps are executed repeatedly. If $N1$ is equal to $N2$, the iteration is terminated immediately. If $N1$ is not equal to $N2$, locus M such that $PA1[M] = PA2[N1]$ is searched. Then, the value of $PA1[N1]$ is substituted to $PA1[M]$, and the value of $PA2[N1]$ is substituted to $PA1[N1]$. RS is used as the index to quickly search locus M by the value of a gene. Then $N1$ is incremented and the above process is repeated until $N1$ becomes equal to $N2$.

After the above process, the chromosome stored in $PA1$ is used

as the new chromosome, and sent to the mutation module. The chromosome in $PA2$ is used as $parent1$ for the next operation. For the purpose, in the next operation, registers $PA1$ and $PA2$ are used as $PA2$ and $PA1$, respectively.

This module takes $\frac{5}{2}n$ clock cycles to generate a new individual, where n cycles are required to load a parent chromosome in $PA2$, $\frac{n}{2}$ cycles in average for the crossover operation, and n cycles for sending the new chromosome to the mutation module. The required clock cycles can be reduced to $\frac{3}{2}n$ by receiving a parent chromosome in $PA2$ from the management module and transmitting the new chromosome from $PA1$ to the mutation module, at the same time.

As we explained in Sect. 4.3, it is impossible to execute the GA pipeline using this crossover module without stall, since the module takes n to $2n$ clock cycles depending on the random numbers $N1$ and $N2$. However, by duplicating the crossover module, we can let the module always take n clock cycles for each crossover operation[1].

Mutation Module

The mutation module receives a chromosome from the crossover module, applies the mutation operation to it, and sends it to the evaluation module. A general mutation operation for TSP is to swap genes at two loci selected at random in the chromosome. Here, we adopted the following approximation of this technique.

First, it generates two random values $N1$ and $N2$ between 0 and $n-1$. When this module receives the a gene whose value is $N1$ from the crossover module, it sends $N2$ to the evaluation module instead of $N1$, and if it receives $N2$, it sends $N1$ instead. The above operation is applied at the probability of the given mutation rate.

It is easy to see that the mutation module approximately takes n clock cycles to process one chromosome.

Evaluation Module

The fitness value of a chromosome is calculated as a total distance to travel all cities in the specified order. We let the evaluation module retain distances between any two cities in the memory, where the distances are initialized from outside before the circuit runs. Each distance is, for example, coded as a 8 bit integer value. A distance between two cities C_1 and C_2 is stored in the address expressed in the following equation in the memory.

$$F\_address(n) \quad = \quad n \times C_1 + C_2, \tag{1}$$

where n is the number of cities. When the evaluation module receives a gene, it fetches the distance between the city of the received gene and the previous one from the memory and adds the distance to the fitness value. Although these operations require three clock cycles, they can be pipelined with the receiving operation of a chromosome. Then the calculated fitness value is sent to the management module. It requires one clock cycle.

Consequently, the evaluation module approximately takes n clock cycles to calculate a fitness value for one chromosome and to send it to the management module.

6. PREDICTION MODEL FOR COST PERFORMANCE TRADEOFFS

As we explained in Sect. 4, our proposed architecture consists of multiple GA pipelines where the communication mechanism among those pipelines is quite simple. So, if the size of a single GA pipeline is known, we can easily estimate the size of the whole circuit for the given number of concurrent GA pipelines. A GA pipeline consists of several modules for GA operations. The circuit size for each module varies depending on the problem size (the number of bits to represent each individual). So, we develop a model for predicting the size of each module.

Here, we put a strong assumption that the circuit size for each module increases in proportion to the logarithm of a given problem size. The actual circuit size of each module can be obtained by synthesizing the circuit. So, our approach is to synthesize modules with different problem sizes, and to obtain linear functions using multiple regression.

We use actually synthesized circuit sizes for each module in multiple regression. As a result, we can obtain a linear function of the logarithm of a problem size to predict the circuit size of each module. For the management module, we predict the circuit size with a linear function of the logarithms of a problem size and a population size (the number of individuals). In order to calculate coefficients of linear functions with multiple regression, we used three different values for each parameter (a problem size and a population size).

When the circuit sizes of all modules are known, the size of the whole circuit can be calculated as the sum of the sizes of modules used in the circuit.

We show the accuracy of the proposed prediction model through experiments in Sect. 8.

7. TOOL TO SUPPORT DESIGN AND IMPLEMENTATION OF GA CIRCUITS

We have developed a tool to automatically generate an RT level VHDL description of the GA circuit designed with our architecture, allowing designers to obtain parameter values, that is, the problem size, the population size, and the number of concurrent GA pipelines, suitable for a specified FPGA device. The tool also allows designers to check whether the specified parameter values can generate the circuit which can be implemented on the target FPGA device in terms of the number of logic elements and the number of memory blocks.

The tool consists of the *circuit size check part* and the *circuit derivation part*. The circuit size check part calculates the maximum number of concurrent pipelines in the circuit from the problem type, the problem size, the population size, and the target FPGA device. The circuit derivation part generates the RT-level VHDL description based on our parallel architecture from the parameter values calculated by the circuit size check part.

The circuit derivation part utilizes a library containing VHDL descriptions of a variety of modules for GA operations, a set of template files required to automatically generate the circuit description, and component files. Each template file includes the library declaration (Fig. 3 (a)), the entity declaration (Fig. 3 (b)) and part of the architecture declaration (Fig. 3 (c)). Each component file includes component instances of the GA circuit and interface modules.

The circuit derivation part generates a circuit description with specified parameter values by substituting the values to variables in the template file (e.g., `%population_size%`) and by rewriting part of the component file.

By preparing VHDL descriptions of GA operations for various problems as a library and the corresponding component files, the proposed tool facilitates easy design and implementation of efficient circuits for target problems and for specified FPGA devices.

For example, the tool can quickly find that Cyclone EP1C12 (12060 LE) [12] can accommodate a GA circuit with two concur-

[1]Note that our experimental results in Sect. 8 does not use the duplication of the module.

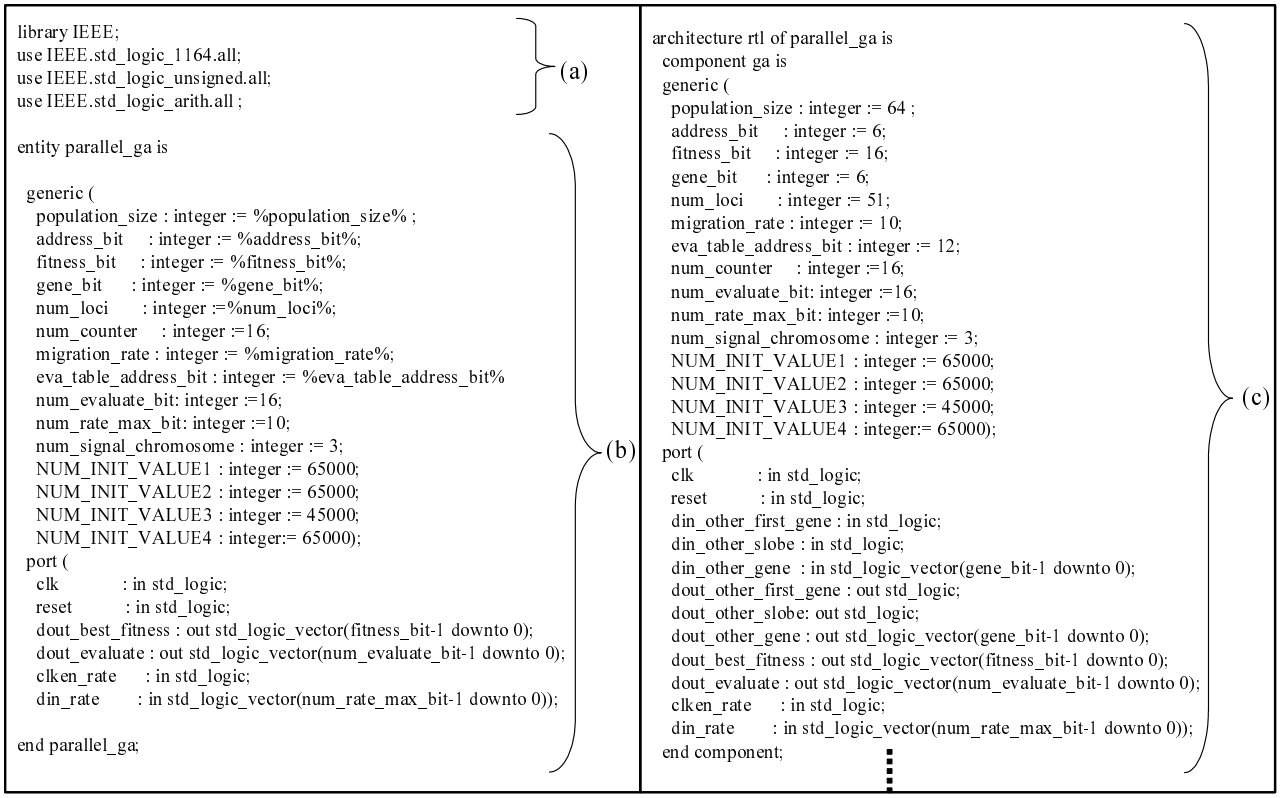


Figure 3: Example for template file.

rent GA pipelines which solves 64 bit Knapsack Problem. It can also generate the VHDL description of such a circuit. The total execution time of our tool is less than 1 second for most cases including the above problem.

8. EXPERIMENTAL RESULTS AND EVALUATION

In this section, we show through experiments the performance of the circuits implemented based on our proposed architecture in Sect. 4 and the accuracy of our prediction model in Sect. 6.

For Knapsack Problem and TSP, we have described RT-level VHDL descriptions of modules for GA operations (see Sect. 5) and generated circuit descriptions with several numbers of concurrent pipelines using our tool in Sect. 7. Then we conducted logic synthesis for the circuit descriptions using Altera Quartus II. We used Altera Cyclone FPGA devices as target devices, and used its internal memory to store individuals and data used by the final circuits.

8.1 Performance Evaluation

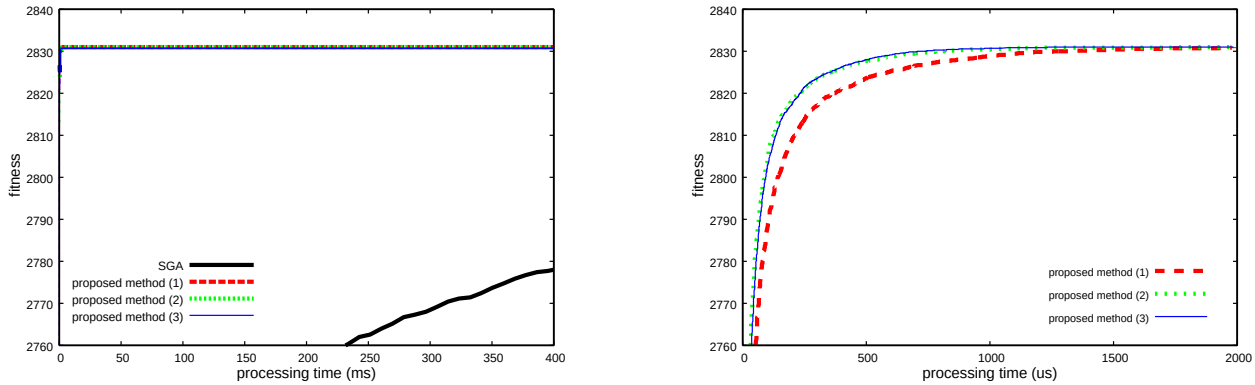
Here, we have conducted two kinds of experiments: one is to investigate superiority of our hardware implementation of GA to software implementation; and the other is to investigate the scalability of the proposed architecture in terms of the number of concurrent pipelines.

In order to evaluate performance of the generated circuits, we investigated the quality of solutions and the search speed. We used 64 bit Knapsack Problem and a TSP instance called eil51, which calculates a semi-optimal route to travel 51 cities. We compared the circuits implemented based on our method with software implementation of GA executed on a PC with Linux OS, Pentium 4 2.4GHz and 256MB memory.

When we implement GA as software, there is less restrictions on the algorithm of the crossover operation and generation model. So, we used the general generation model of GA (see Sect. 3) in software implementation for both Knapsack Problem and TSP. Hereafter, we refer to the software implementation as *SGA*, that is the abbreviation of the *simple genetic algorithm*. For TSP, we used the complex but efficient crossover algorithm called EXX [13] in SGA. We also implemented another SGA using PMX (see Sect. 5) to see difference between those two crossover algorithms. We used gcc version 2.95.4 to compile programs for SGA, and specified `-O3` as the optimization option.

At first, we measured how good solutions can be obtained using the same execution time, for our circuits and SGAs. Here, the execution time of our circuits on FPGA is calculated by simulation on Quartus II.

The results are shown in Fig.4, and 5. Fig. 4 (b) is the magnification of Fig. 4 (a). In Fig. 4 (a) and (b), the higher fitness value means the better solution. In Fig. 5, the lower fitness value means the better solution. The number in parentheses such as `proposed method (3)` indicates the number of concurrent pipelines in the circuit. In Fig. 5, `proposed method (8)` and `proposed method (16)` could not be implemented on a single FPGA device. These are presented to show the scalability of the proposed method. We suppose that these circuits work at the same clock



(a) (b)

Figure 4: Search efficiency for Knapsack Problem

Table 1: Processing time per one individual.

Problem	Problem Size	Basic Architecture	SGA
Knapsack Problem	16	7.09 (ns)	0.508 (μs)
	32	7.25 (ns)	0.765 (μs)
	64	9.26 (ns)	1.36 (μs)
	128	8.47 (ns)	2.24 (μs)
TSP	51	1.28 (μs)	2.07 (μs)
	76	2.06 (μs)	2.25 (μs)
	101	2.99 (μs)	3.32 (μs)

speed as `proposed method (4)`.

Fig. 4 shows that our circuits achieve much better performance than SGAs, for Knapsack Problem. Fig. 4 (b) shows that our circuits calculate good solutions in short time and the larger number of concurrent pipelines slightly contributes the improvement of the performance. This is because Knapsack Problem is too simple for parallel implementation. Fig. 5 shows that in our circuits candidate solutions converge to semi-optimal ones much more quickly than SGAs and that the quality of solutions in our circuits is comparable to that of SGA (EXX) which uses the better crossover algorithm. Also the figure suggests us that the performance of our circuits is greatly improved by increasing the number of GA pipelines. When giving enough execution time, the quality of solutions becomes better in SGA with EXX than our circuits. However, if we implement EXX in our circuits, the quality of solutions would greatly be improved, although the circuit size would much increase as well. This is a tradeoff.

Finally, we show the operation time per individual and approximate power consumption in Table 1 and Table 2, respectively. Table 1 shows that the generated circuit has much shorter execution time per individual than SGAs. Table 2 shows that the generated circuit consumes at most 1/80 of TDP (Thermal Design Power) for Pentium 4 2.4GHz.

8.2 Evaluation of Prediction Model

We measured the sizes of the synthesized circuits for different parameter values such as the number of concurrent GA pipelines, the problem size, and the number of individuals. We also predicted the sizes using our prediction model and evaluated its accuracy by comparing the predicted sizes and the actually synthesized circuit sizes.

Comparison between predicted size and actual size of each circuit is shown in Fig. 6. Here, each number in parenthesis in these figures represents the number of concurrent pipelines in the corresponding circuit. In the figures, sizes of the interface circuits are also calculated using our prediction model. According to the results in Fig. 6, we see that prediction error is within 3% for the maximum for both Knapsack Problem and TSP.

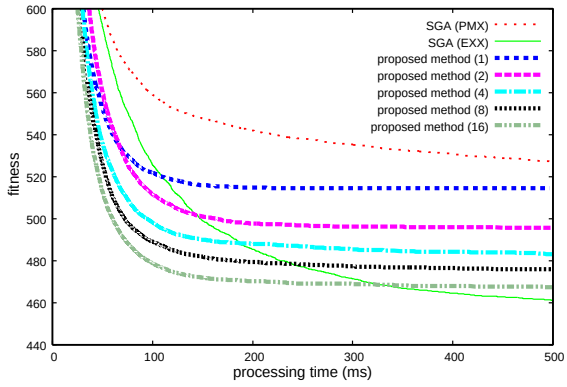


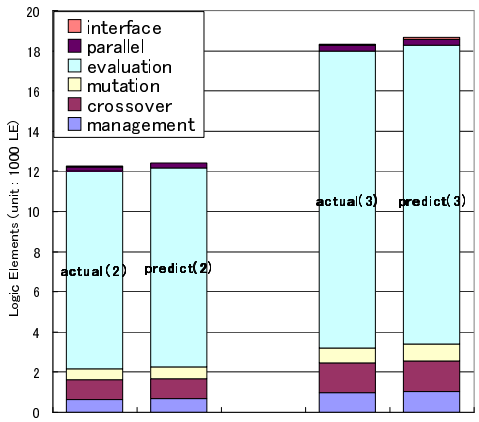
Figure 5: Search efficiency for TSP(eil51).

Table 2: Power Consumption of Basic Architecture.

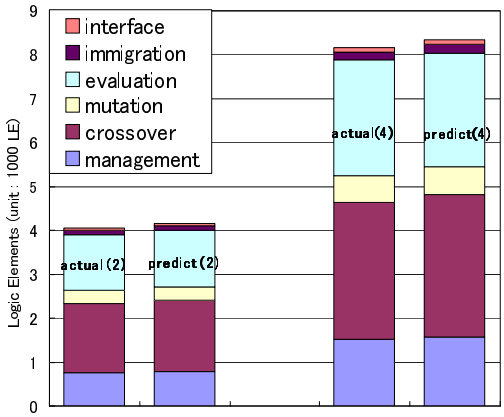
Problem	Device	Total Power
-	Pentium4(2.4GHz)[14]	57.8 (W)
Knapsack Problem	FPGA($s = 16$)	293 (mW)
	FPGA($s = 32$)	362 (mW)
	FPGA($s = 64$)	427 (mW)
	FPGA($s = 128$)	700 (mW)
TSP	FPGA($s = 51$)	476 (mW)
	FPGA($s = 76$)	511 (mW)
	FPGA($s = 101$)	611 (mW)

9. CONCLUSION

In this paper, we proposed a flexible hardware implementation technique for GAs on FPGAs, aiming mainly at embedding FPGAs on portable computing devices and/or information appliances to enhance computation power with low cost and low power consumption. With the proposed method, we can easily prototype the



(a) Knapsack Problem, 64 items



(b) TSP, eil51

Figure 6: Accuracy of Prediction Model

suitable hardware architecture with multiple concurrent pipelines depending on the problem size and the size of the target FPGA device. Through experiments, we confirmed that our circuit achieves the performance much higher than Pentium4 2.4GHz with less than 1/80 power consumption, and that our prediction model can predict the circuit size within 3 % error. We think these results are good enough for practical use.

Throughout the paper, we used Knapsack Problem and TSP as target problems solved by GAs. However, our proposed architecture is applicable to various GA-based problems, and our tool would be much more useful if the RT level circuit descriptions for GA operations of various problems are prepared as a library. As part of future work, we would like to design a high-level language to specify algorithms for GA operations so that we can generate the corresponding RT-level hardware descriptions automatically. It will also be our future work to extend our prediction model to be able to predict power consumption in the synthesized circuits.

10. REFERENCES

[1] Atsushi Maruyama, Naoki Shibata, Yoshihiro Murata, Keiichi Yasumoto and Minoru Ito, P-TOUR: A PERSONAL NAVIGATION SYSTEM FOR TOURISM, Proc. of 11th World Congress on Intelligent Transport Systems, 2004.

[2] Li Layuan, Li Chunlin, Genetic Algorithm-Based QoS Multicast Routing for Uncertainty in Network Parameters , Web Technologies and Applications, 5th Asian-Pacific Web Conference (APWeb 2003), pp. 430–441 , 2003.

[3] Hidenori Sakanashi, Masaya Iwata, and Tetsuya Higuchi, Lossless Compression of Very High Resolution Bi-level Images Using Genetic Algorithm, Journal of Information Processing Society of Japan, vol. 45, No. 5, pp. 1460–1470, 2004 (in Japanese).

[4] Paul Graham and Brent Nelson, A Hardware Genetic Algorithm for the Traveling Salesman Problem on SPLASH 2, Field-Programmable Logic and Applications, pp. 352 – 361, 1995.

[5] 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.

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

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

[8] 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,Proc.RTSS2004, pp. 437-446, 2004.

[9] 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.

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

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

[12] Altera Corp, Cyclone Device Family Data Sheet, `http://www.altera.com/literature/hb/cyc/cyc_c5v1_01.pdf`.

[13] Keiji Maekawa, Hisashi Tamaki, Hajime Kita and Yoshikazu Nishikawa, A Method for the Traveling Salesman Problem Based on the Genetic Algorithm, Transactions of the Society of Instrument and Control Engineers , vol. 31, No 5,pp.598–605,1995 (In Japanese).

[14] Processor Spec Finder, `http://processorfinder.intel.com/scripts/default.asp`.