The Hardware Design for a Genetic Algorithm Accelerator for Packet Scheduling Problems

Yang-Han Lee[1]*, Yih-Guang Jan[1], Yun-Hsih Chou[2], Hsien-Wei Tseng[1], Ming-Hsueh Chuang[1], Shiann-Tsong Sheu[3], Yue-Ru Chuang[1], Jei-Jung Shen[1] and Chun-Chieh Fan[4]

[1]*Department of Electrical Engineering, Tamkang University, Tamsui, Taiwan 251,R.O.C.*
[2]*Department of Electronic Engineering, St. John's University, Tamsui, Taiwan 251, R.O.C.*
[3]*Department of Communication Engineering, National Central University, Taoyuan, Taiwan 320, R.O.C.*
[4]*Department of Computer & Communication Engineering, St. John's University, Tamsui, Taiwan 251, R.O.C.*

Abstract

In the basic genetic algorithm and its variations, they usually process the calculations in a sequential way so that the waiting time for every generation member awaited to be processed increases dramatically when the generation evolution continues. Consequently the algorithm converging rate becomes a serious problem when we try to apply the genetic algorithm in real time system operations such as in the packet scheduling and channels assignment in the fiber optic networks. We first propose in this paper a genetic algorithm accelerator which has the capability not only to accelerate the algorithm convergent rate but also to have its solution to reach the problem's optimum solution. Then we develop hardware blocks such as the blocks of Base Generator, Operation Selector, Delta Calculator, Duplicate Priority Encoder, Abort Priority Encoder and Next Generator, etc. to realize this proposed generic algorithm accelerator. Due to these hardware blocks realizations it will enhance the speed of the algorithm converging rate and make certain its convergent solution reaches the problem's optimum solution.

***Key Words*:** Genetic Algorithm, Packet Scheduling, Base Generator, Operation Selector, Delta Calculator, Duplicate Priority Encoder, Abort Priority Encoder, Next Generator

1. Introduction

From the point of view of Darwin's theory of evolution, those organisms suitably for living will survive through many evolutions. From many experimental tests it demonstrates that genetic algorithm is not only an effective but also an efficient method for searching the problem's optimum solution. Its applications cover a wide range of areas such as in the field of needing an enormous data computations, information retrieval, timing closure, packet scheduling and real time system operations etc.

In the genetic algorithm, it uses the same philosophy as in the evolution theory, the organisms use special rules to combine and arrange many genes to form their chromosomes. The genetic algorithm considers every possible solution considered as a linear chromosome set which consists of several serial problem parameters for the problem considered. It then uses binary encoding to digitize every chromosome to form an information code for chromosome. By observing these information codes we can examine in the simulation environment every chro-

*Corresponding author. E-mail: yhlee@ee.tku.edu.tw

mosome behavior. By implementing crossover, mutation and selection processes it continuously generate new generations with better qualities and eliminate those inferior chromosomes so as to reach the problem's optimum solution [1].

In traditional genetic algorithm it needs to process the calculation sequentially for each generation member and the computation complexities increase as the chromosomes evolve from generation to generation. It results that the waiting time to process every generation member increases dramatically. The evolution speed or the converging speed, i.e. the time from starting the evolution till finding the optimum solution, becomes slower and it needs to wait an enormous time for their turn to be processed. Furthermore the system needs a lot of time to process the codes matching and mutation operations so the evolution time for every generation will be dragged accordingly. It therefore does not have any improvement in the converging speed by only digitizing the chromosomes. It therefore could not satisfy the high performance real time operation requirement. It then becomes a challenging issue for the industry and the program developers to find ways to speed the converging rate in the genetic algorithm operation to avoid the long cumbersome computation time and even further to consider the way to meet the real time system [2–4] requirement when information is transmitted through the optic fiber network [5].

An example, as shown in Figure 1 [7], is given to illustrate the scheduling problem in a star-based network that a number of packets with variable lengths from four nodes (N) arriving at Passive Star Cobbler (PSC), in which there are K parallel channels per fiber (in usual, the number of nodes is smaller than the number of wavelengths.). In this figure, notation P_{ij} is denoted as the j-th packet from node i. In order to minimize the total packet switching delay time and maximize the channel utilization, these packets should be well scheduled in these K available channels. In the literatures, the scheduling of sequencing tasks for multiprocessor has been addressed extensively and proved to be an NP-hard problem [6]. Similarly, the packet scheduling and wavelength assignment problem under the constraint of maintaining the sequence in order is also well known as a difficult-to-solve issue. From searching available literatures, we believe that it is hard to design a real-time scheduling algorithm to resolve the NP-hard problem by implementing general heuristic schemes. The organization of this paper is as follows. In Section 2 we introduce the general genetic algorithms for packet scheduler so as to introduce the definition of general genetic algorithm, the usage of fitness function and its implementation. In Section 3 we introduce the hardware design for the parallel-processing of the genetic algorithm and then we discuss in detail of each hardware function block in Section 4. We draw conclusions in Section 5.

2. General Genetic Algorithms for Packet Scheduler

The G-GA, General Genetic Algorithm, mechanism applied in industrial engineering contains three main components: the Crossover Component (XOC), the Mutation Component (MTC) and the Selection Component (SLC), as shown in Figure 2 [7,8]. A process to apply the G-GA mechanisms in solving the optimization problem of packet scheduling and wavelength assignment in networks is named as the General GAs Packet Scheduler

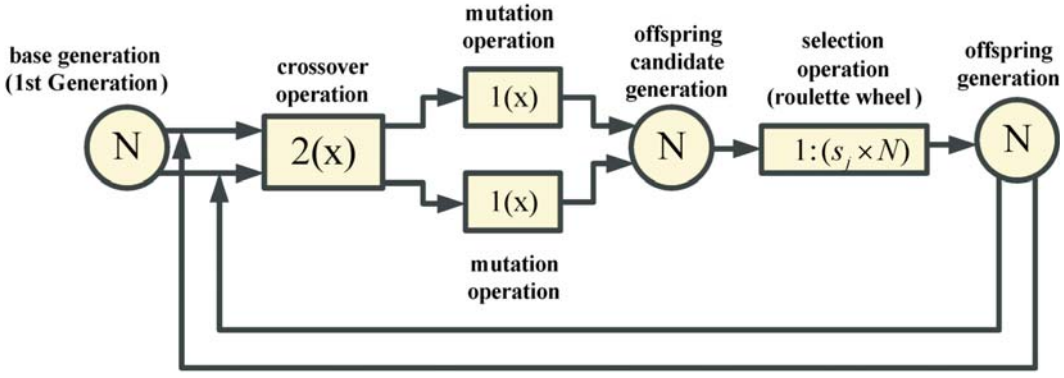

Figure 1. An example illustrates the packet scheduling problem in a star-based network.

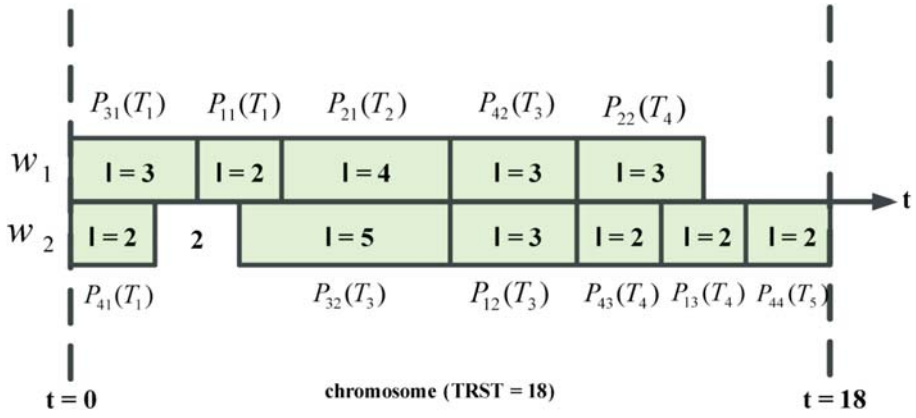

Figure 2. The flow block diagram of the G-GAPS.

(G-GAPS) [7]. Basically, the G-GAPS needs to use a collection window (*C*) for collecting packets. As soon as the scheduling process is executed, a new collection window will be started. By using this collection window, it will smooth the traffic flow if the window size is properly selected.

2.1 Definition

In G-GAPS, packets destined to the same output port are collected and permutated for all available wavelengths to form a *chromosome* (i.e., a chromosome presents a kind of permutations), in which each packet is referred to a *gene*. The example shown in Figure 3 [7] demonstrates that a set of collected packets (*P*) with different lengths (*l*) and time stamps (T) are permutated for two available wavelengths (*W1* and *W2*) to form a chromosome. A number of chromosomes, denoted as *N*, will be first generated to form the *base generation* (also called the *first generation*). Following the sequencing, each arrival packet is associated with a time stamp and all permutations will follow these time stamps of packets to be executed as in the scheduling principle. Therefore, the problem becomes of finding the proper switching time and to decide the associated wavelength of a packet so that the precedence relations of the same connection can be maintained and the total required switching time (TRST) of the schedule can be minimized. More definitely, the TRST presents the maximum scheduled queue length of packets assigned into different wavelengths. Therefore, we can define the TRST(*j*) by the formula as [7]:

$$TRST(j) = \max\{trst(w_j(1)), trst(w_j(2)),....., trst(w_j(K))\} \quad (1)$$

where *j* is the *j*-th chromosome, and trst($W_j(k)$) is the TRST for those packets scheduled in the *k*-th wavelength of the *j*-th chromosome. Here we assume an opti-

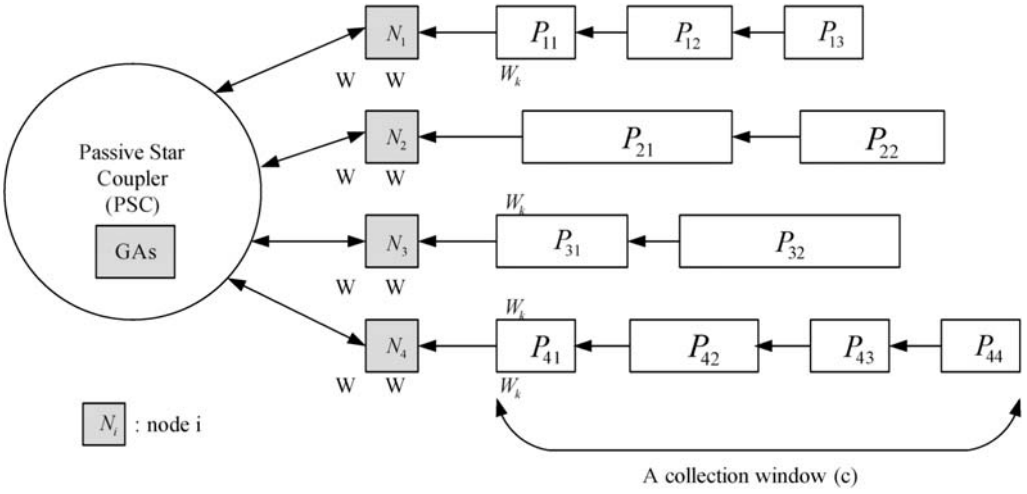

Figure 3. An example presents a permutation to form a chromosome.

cal fiber carries *K* wavelengths.

2.2 The Fitness Function

The *fitness function*, denoted as Ω in the G-GAPS is defined as the *objective function* that we want to optimize. It is used to evaluate various chromosomes during selection operation to determine which offspring should be remained as the parents for the next generation. The objective function in the scheduling is the TRST and it is often converted into maximization form. Thus, the fitness value of the *j*-th chromosome, denoted as Ω (*j*), is calculated as following [7]:

$$\Omega(j) = \Psi_1^{worst} - TRST(j) \quad (2)$$

Where $\Psi_1^{worst} = \Sigma_u \Sigma_v l_{uv}$ represents the worst TRST in the first generation (i.e., all packets are scheduled in one wavelength.). Therefore, the optimal schedule will be the chromosome with the largest fitness value denoted as Ω^{opt}.

2.3 Implementation of the Genetic Algorithms

In G-GAPS, each crossover operation selects two chromosomes from the same generation and generates two new offspring that treated as the candidates for the next generation. These candidate offspring will involve in the mutation operation and in the selection operation according to their mutation probabilities (*Pm*) and fitness values, respectively [7].

In the implementation, we simply assume the number of chromosomes in the base generation, say *N*, is even. Let *Pc* and *Pm* denote as the crossover and mutation probabilities, respectively. According to roulette wheel method, the probability of selecting the *j*-th chromosome is $s_j = \Omega(j) / \Sigma_{r=1}^{r=N} \Omega(r)$.

3. Hardware Design for the Genetic Algorithm Accelerator

When problems, for whatever reasons, could not be solved by common mathematical operations it is usually resorted to using computer computations. Hardware for genetic algorithm accelerator [8] has been developed in the computation algorithm to speed its computation so as to quickly converge to a result that is also close to the optimal solution for the problem. The accelerator consists of (1) a chromosome generator to generate the initial population of chromosomes that each chromosome has distinct information code, (2) a chromosome accumulator in which there is at least a crossover unit and many mutation units to crossover son generation chromosomes that mostly with different codes. It is then, according to the evolution process for the generation of these chromosomes, to develop and evaluate the functional relation of Fitness Value for this son generation chromosomes, (3) an Offspring Candidate Pool to collect and group those son generation chromosomes that meet the requirements, i.e. to have a value exceeds a standard adaptive value, and (4) an Offspring Pool to select those candidate chromosomes in each group suitable for crossover and store them into the Offspring Pool. In these processes executions, they can meet high speed real time computation requirement because they are all designed and processed in the architecture so as to increase their processing speed. By extending the base architecture as discussed above we have the hardware architecture for the genetic algorithm accelerator as shown in Figure 4. The IO registers list and their descriptions are shown in Table 1.

4. The Description of Functional Blocks Description

4.1 Random Generator

One of the most important issues of implementing Genetic Algorithm is the random probability. To make the whole algorithm like an "Natural Selection" we design an 89th order generator polynomial to generate maximal length (M-sequence) PN codes orders to simulate the corresponding random numbers. The octal representation of the generator polynomial is [400,000,000, 000,000,000,000,000,000,151], and the random generator will start to shift when the power is on and we take and assign part of the bits of the random generator to the corresponding blocks to assume the roles of the random numbers when the Genetic Algorithm is operating.

4.2 Base Generator

An architecture of the base generator is shown in Figure 5. We can choose the random number or the IO register setting value to be the first generation chromosome by setting the IO register "base_sel". If users have better first generation chromosome values from prior-calculation, or previous result or experience values, they

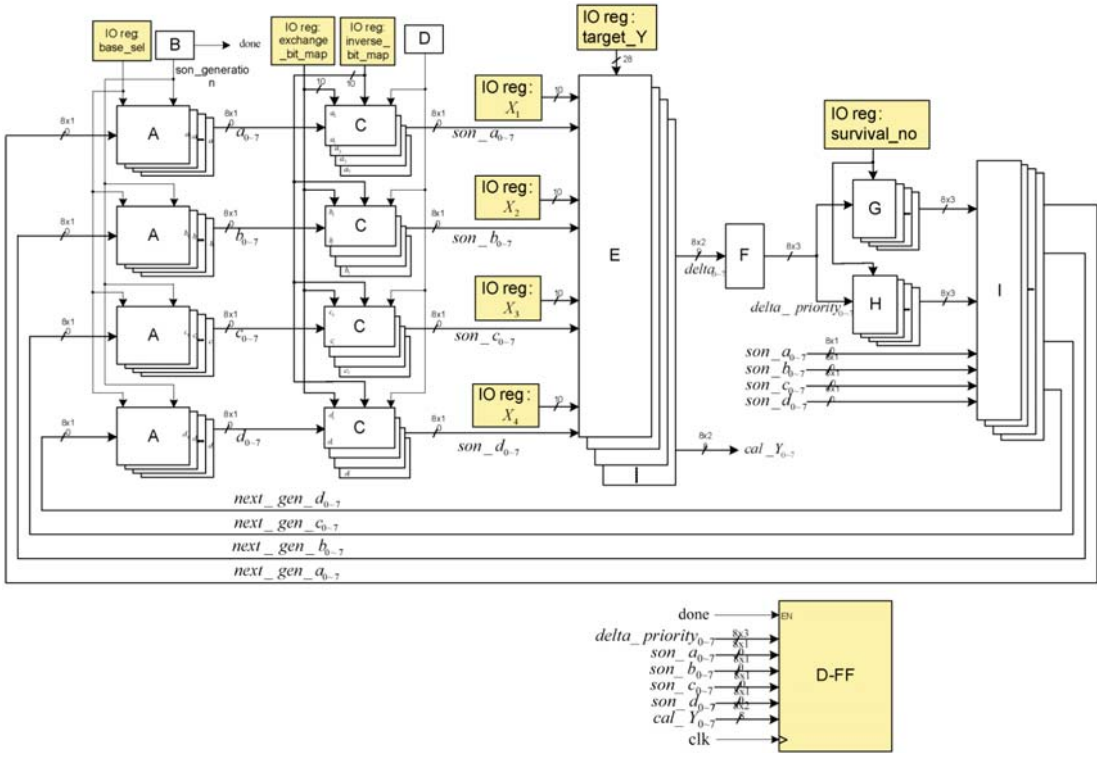

Figure 4. Hardware architecture for genetic algorithm accelerator.

Table 1. IO registers list and their descriptions

base_sel:	base generation source selection 0: from random generator 1:user-defined as in the following IO registers
base_a0[9:0] ~ base_a7[9:0]	user-defined base generation a0~a7
base_b0[9:0] ~ base_b7[9:0]	user-defined base generation b0~b7
base_c0[9:0] ~ base_c7[9:0]	user-defined base generation c0~b7
base_d0[9:0] ~ base_d7[9:0]	user-defined base generation d0~b7
total_gen_no[4:0]	total operating generation numbers
exchange_bit_map[9:0]	define which bits supposed to be exchanged in crossover
inverse_bit_map[9:0]	define which bits supposed to be inversed in mutation
compare_seed[3:0]	if the random number equals to the compare seed, the mutation will be done in this generation
compare_mask[3:0]	choose which bits would not be compared with the compare seed to determine the mutation probability
X1[15:0] ~ X4[15:0]	the input values of the fitness value formula
target_Y[27:0]	the target fitness value of the fitness value formula
survival_no[2:0]	determine the survival numbers of the son chromosomes to pass to next generation

can fill '1' to the IO register "base_sel" after setting the sets of IO registers "a0~a7,b0~b7,c0~c7,d0~d7" or fill '0' to the IO register "base_sel" to load the random numbers from the random generator. There is a multiplexer block to determine the first generation chromosome or the son of the previous generation to enter the following calculations by the signal "son_generation" from the block B "Generation Counter". There is a set of the only set and is the only set of flip-flops in the calculation cycle, to latch the current operators to the following calculation.

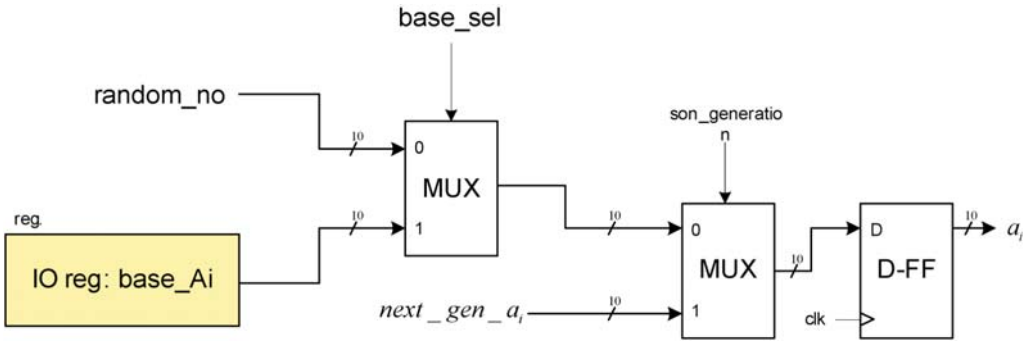

Figure 5. Base generator.

4.3 Generation Counter

A generation counter is shown in Figure 6. This block records the current generation and compares it with the value of IO register "total_gen_no" to determine is it the correct number of generations to stop the procedure and then to store the final estimating parameters from the fitness value formula. If the counting value is '0', it means it is the base generation. If the counting value is not '0', it means it is the son generation and the signal "son_generation" will go to '1' to inform the block A "Base Generator" to pass the son chromosomes of last generation through multiplexer to the next calculation cycle. If the value of the generation counter equals to total_gen_no, the signal "done" will go to '1' to informs that it is the final generation and to command the resulting flip-flops to store the final estimating parameters from the fitness value formula. On the other words, users can control the total generation numbers by setting the IO register "total_gen_no".

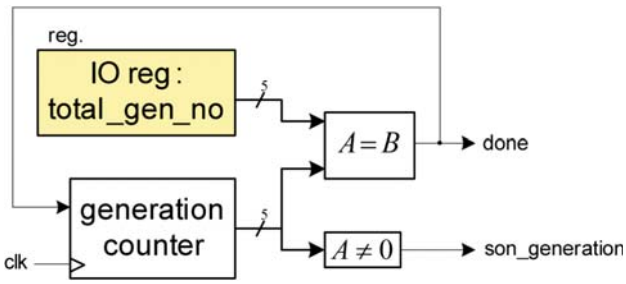

Figure 6. Generation counter.

4.4 Operator

The operation is shown in Figure 7. This block plays the role of "God". It imitates the life's crossover and mutation of genes to generate and different possible operators so as to increase the possibility of finding the optimal fitness value. It contains 3 sub-blocks, the crossover, the mutation and the multiplexer.

● Crossover:

In Figure 8, it shows the operation of the crossover. It

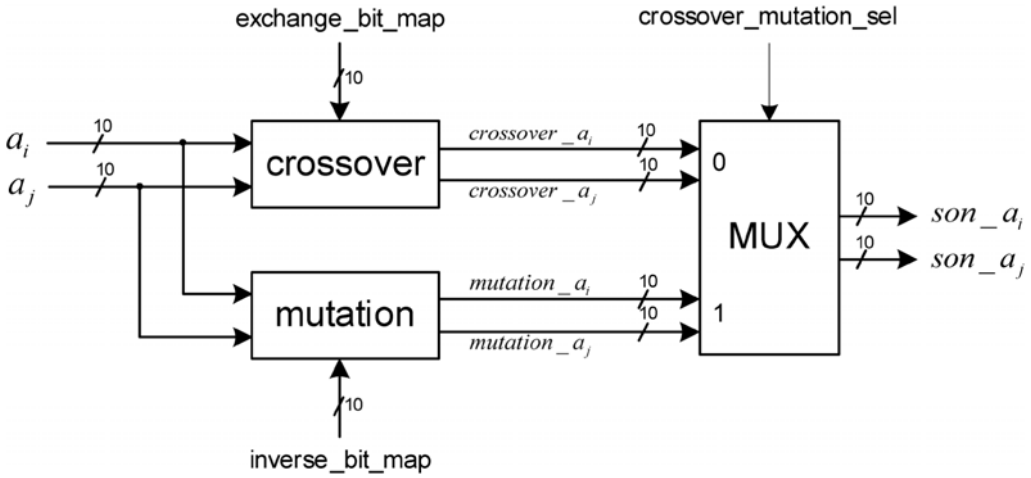

Figure 7. Operator.

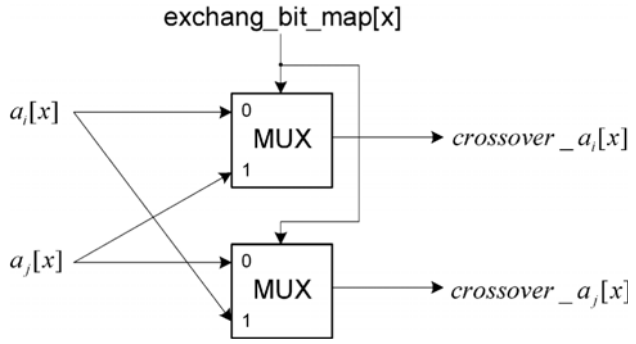

Figure 8. Crossover.

references to the IO register "exchange_bit_map" to exchange the bits of two different genes in the same group to create the various different son generation. For example if the IO register "exchange_bit_map" equals 10'b01, 0000,1100, the crossover of "a" group will exchange bit 8, bit 3 and bit 2 of a0 and a1 to create different son generation.

● Mutation:

The mutation operation is shown in Figure 9. It references to the IO register "inverse_bit_map" to inverse the bits of the input gene to create another kind of son generation. For example if the IO register "inverse_bit_map" equals 10'b01,0000,1100, then the bit 8, bit 3 and bit 2 of the input genes will be inversed by the mutation.

● MUX:

This multiplexer will reference to the signal "crossover_mutation_sel" of the block D "Operation Selector" to pass the crossovered genes or the mutated genes to the following operation.

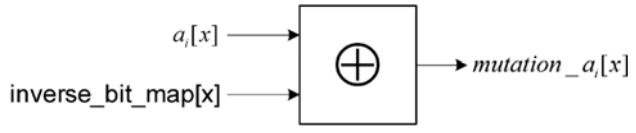

Figure 9. Mutation.

4.5 Operation Selector

As shown in Figure 10 is the block of operation selection. This block is to determine which operation will be processed in this generation, either crossover or mutation. Before this we should set two IO registers "compare_seed" and "compare_mask". This block will compare the "compare_seed" with the random number generated from part of the bits of the random generator, and if they are match, it means this generation will be in mute operation otherwise this generation will be in crossover operation. But because the seed is a 4-bit number, the probability that the seed and the random number are the same is fixed at 1/16, and we don't like this kind of fixed probability, so we add a IO register "compare_mask". If the corresponding bit of the "compare_mask" is '0', we will compare this bit. On the other hand, if the corresponding bit is '1', we will not compare that bit and think they are the same. Now we can change the probability of the mutation by setting the IO register "compare_mask". If the "compare_mask" contains two '1s', it means the probability of the mutation is 1/4.

4.6 Fitness Value and Delta Calculator

In Figure 11, it shows the functional operation of the fitness value and the delta calculation. After a total operations of 32 parallel "Operator" blocks, we have collected all the new son generations and now we can calculate the fitness value by using new eight sets of son operators individually. In this block we take the inputs of value X1~X4 and set in the IO register "X1~X4" positions and use the new generation genes to calculate the corresponding "cal_Y" value, and then compare it with the "target_Y" in the IO register to find their difference "delta".

4.7 Delta Compare and Encode

Delta compare and encode functional block is shown

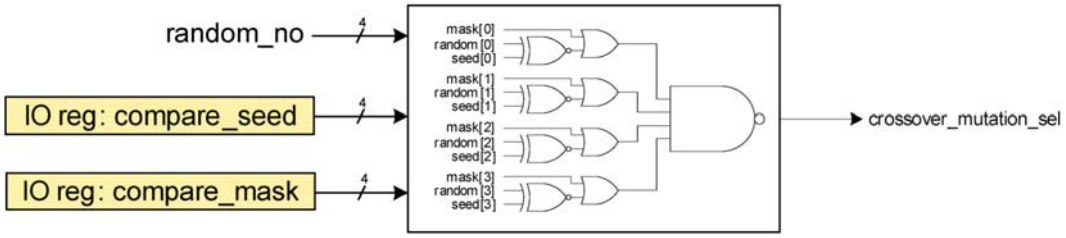

Figure 10. Operation selector.

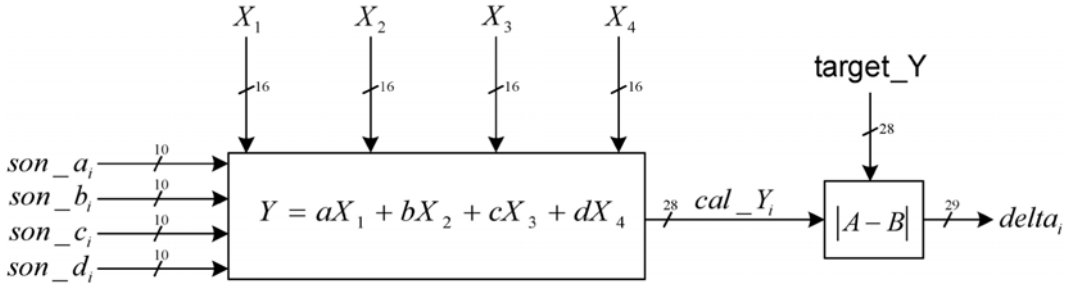

Figure 11. Fitness value and delta calculator.

in Figure 12. After the block "Fitness Value and Delta Calculator", we get the difference value between the calculated and the target fitness values of the eight sets of new genes. To further analyze, we compare each difference "delta" and give each of them a priority named "delta_priority". If the value of "delta" is small, it means that the set of genes is closer to our target and that is a good set of genes, and we give it a higher priority. On the other hand, the value of the priority is low.

4.8 Duplicate Priority Encode

Duplicate priority encode functional block is shown in Figure 13. We have sorted the eight sets of new genes and make out them from the best to the worst, and we have to find out how many and which should be duplicated to replace the bed genes. In this block, we can reference to the IO register "survival_no" to determine how many genes should be duplicated and the value in the "delta_priority" to give each one a "duplicate_priority". If the delta priority is greater than the survival number, it means that it should be duplicated to replace the bad gene, and if the "delta_priority" is equal to or less than the survival number, it means that it should be duplicated to replace others, and we set its "duplicate_priority" to '0'.

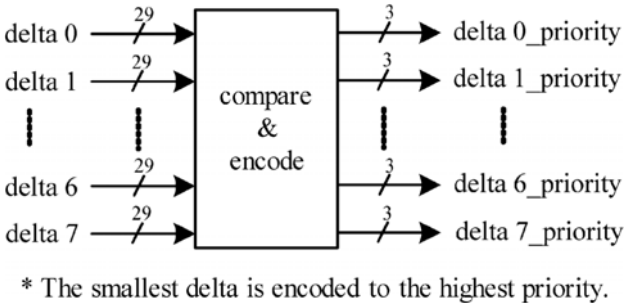

Figure 12. Delta compare and encode.

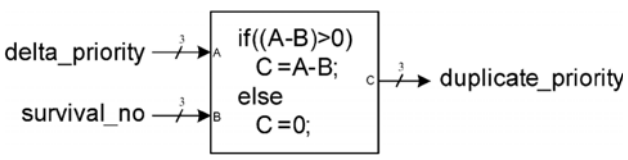

Figure 13. Duplicate priority encode.

4.9 Abort Priority Encode

The abort priority block is shown in Figure 14. The operation of this block is like the above block G, but this block is to find out the abort priority to determine how many and which sets of genes that should be replaced. If we find the sets of genes should not be aborted then we give it an "abort_priority" value '0'. On the other hand, the worst set of genes has the highest "abort_priority", and the value of the highest "abort_priority" should be equal to the value of the highest "duplicate_priority" because they are all reference to the "survival_no" to determine the "abort_priority" and "duplicate_priority". And if the "abort_priority" is not '0', it should be aborted.

4.10 Ext Generation

The exit generation is shown in Figure 15. Finally, we know which sets of genes should be duplicated to re-

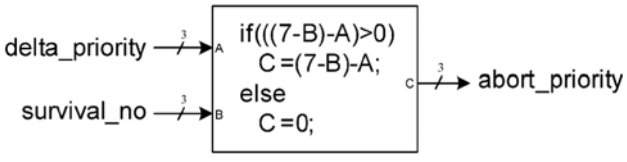

Figure 14. Abort priority encode.

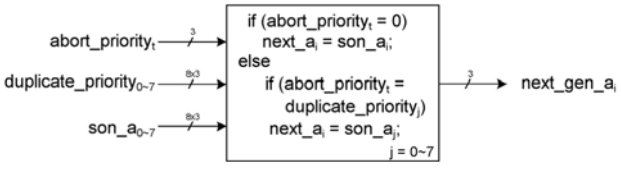

Figure 15. Next generation.

place which sets of genes, and what we have to do is to take the sets of genes which have not '0' "duplicate_priority" to replace the sets of genes which have not '0' "abort_priority" and the value is the same with "duplicate_priority". After we complete all computations of this generation it will generate the new and better genes for the next generation.

5. Matlab Simulation Results

In order to verify that our proposed architecture can generate a better solution, we use Matlab simulation software to verify the excellence of this structure. We use Poisson distribution to determine the packet outcome probability. We use exponential distribution to determine packet length, using 1ms as the smallest unit and setting collection window as 20 ms, Crossover rate 80%, and Mutation rate 5%. At the same time, the simulation of GA by using the conventional architecture, as shown in Figure 1, is processed. We conduct crossover and mutation of chromosomes (packet data), similar to the optimal packet scheduling architecture. However, the largest difference is that the entire packets in one generation must all be processed before the system can proceed on to the next generation. By using to these parameters, the converging patterns, between these two architectures, are shown in Figure 16.

The solid line and the dash lines in the plot represent the Genetic Algorithm Accelerator and G-GAPS, respectively. It is obvious that the new architecture not only has the outstanding result in the first attempt, but also has faster convergence.

6. Conclusion

We developed a genetic algorithm by combining the evolution principal in the traditional genetic algorithm of 'selecting and duplicating the better quality chromosomes for the next generation from large samples of the current generation' and the concept in the steady genetic algorithm of 'reducing computation time between generations to minimize the computation waiting time for codes matching and mutation operations'. Therefore from the real time system point of view and based on the same number of time frames it will generate 'more' and 'better' chromosomes in the genetic algorithm than those in the traditional genetic algorithm. It will speed the converging rate to reach the optimum solution. By realizing this algorithm concept in hardware implementation it further pronounces its real time characteristic.

On the other hand in solving many real time system problems it shows its superior characteristic of using hardware implementation to realize algorithms. Comparing with the simulated results by using the traditional algorithm, the steady state genetic algorithm and the algorithm developed in this paper clearly show their superior performance in solving this kind of real time system problems.

Acknowledgement

The authors would like to thank the National Science Council, R.O.C. for the financial support under Contract NSC 95-2745-E-032-003-URD, NSC 95-2745-E-032-002-URD, NSC 95-2221-E-032-028, NSC 95-2221-E-

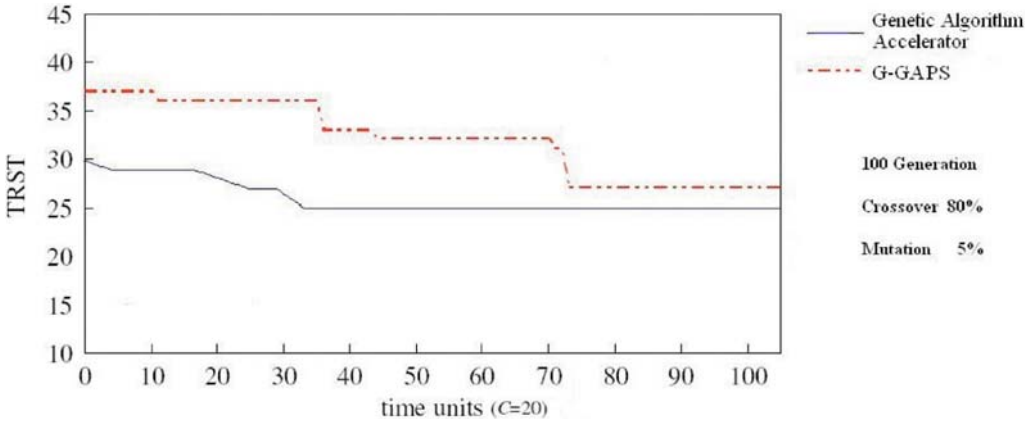

Figure 16. Simulation result of packet scheduling using Matlab software.

032-020, and the funding from Tamkang Unversity for the University-Department joint research project.

References

[1] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, Reading, MA (1989).

[2] Lee, J. H. and Un, C. K., "Dynamic Scheduling Protocol for Variable-Sized Messages in A WDM-based Local Network," *J. Lightwave Technol.,* pp. 1595–1600 (1996).

[3] Babak Hamidzadeh, Ma Maode and Mounir Hamdi, "E_cient Sequencing Techniques for Variable-Length Messages in WDM Network," *J. Lightwave Tech.,* Vol. 17, pp. 1309–1319 (1999).

[4] Sengupta, S. and Ramamurthy, R, "From Network Design to Dynamic Provisioning and Restoration in Optical Cross-connect Mesh Networks: An Architectural and Algorithmic Overview," *IEEE Network,* Vol. 15, pp. 46–54 (2001).

[5] Paul Green, "Progress in Optical Networking," *IEEE Communications Magazine,* Vol. 39, pp. 54–61 (2001).

[6] Hou, Edwin S. H., Nirwan Ansari and Hong Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transaction on Parallel and Distributed Systems,* Vol. 5 (1994).

[7] Sheu, S.-T., Chuang, Y.-R., Cheng, Y.-J. and Tseng, H.-W., "A Novel Optical IP Router Architecture for WDM Networks," *Proceedings of IEEE ICOIN-15,* pp. 335–340 (2001).

[8] Sheu, S.-T. and Chuang, U.-J., "An Optimization Solution for Packet Scheduling: A Pipeline-Based Genetic Algorithm Accelerator," *Proc. Of AAAI GECCO' 2003,* Chicago (2003).

Manuscript Received: May. 5, 2005
Accepted: Jun. 18, 2007