VLSI Hardware Design for Genetic Algorithms and Its Parallel and Distributed Extensions

Norihiko Yoshida [1)] *Tomohiro Yasuoka* [2)] *Toshiomi Moriki* [3)] *Toshihiko Shimokawa* [4)]

1) Department of Computer and Information Sciences, Nagasaki University, Nagasaki 852-8521, Japan
2) Algorithm Research Center, Sony Corporation, Shinagawa, Tokyo 141-0001, Japan
3) Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo 185-8601, Japan
4) Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka 812-8581, Japan

Abstract: The advance in VLSI technologies has enabled us to implement genetic algorithms (GA) in VLSI hardware with a view to drastically improving performance. This paper presents a VLSI hardware design for GA, which we name GAP (Genetic Algorithm Processor), and its extensions for parallel and distributed GA. The basic architecture of GAP employs the steady-state GA, and introduces the simplified tournament selection scheme. This architecture enabled us to achieve two-level parallelization of parallel GA (fine-grained parallelism) and distributed GA (coarse-grained parallelism). We implemented the design of GAP, and evaluated it by logic simulation and logic synthesis. Our examples for evaluation include a data-partitioning problem which is one of the most complex problems ever addressed by GA-VLSIs. Our prototype implementations and experiments prove that the basic architecture of GAP facilitates two-level parallelization of parallel GA and distributed GA, which is effective in improving performance and convergence.

1. Introduction

To obtain drastic performance improvement, it is effective to implement genetic algorithms (GA) in VLSI hardware. The structure of GA computation and problem representation forms a good basis for VLSI hardware. There has already been research carried out on VLSI hardware for GA, some of which concerned specific problems such as pattern matching [1], scheduling [2], the traveling salesman problem [3] and image filtering [4], and some of which are problem-independent [5][6][7][8].

We have been designing VLSI hardware for GA, which we name "GAP (Genetic Algorithm Processor)" [9][10][11]. (1) GAP is a general-purpose (problem-independent) GA-VLSI. (2) GAP employs the steady-state GA, and enjoys efficient pipeline processing. (3) GAP introduces the "simplified tournament selection" for its selection scheme; it is simpler and faster, yet exhibits almost the same convergence compared to the roulette wheel selection which is used in most other GA-VLSIs.

This paper first summarizes the design of GAP, and the advantages of GAP over other GA-VLSIs. We then present extensions to GAP for parallel and distributed GA. "Multi-GAP", i.e. GAP with these extensions, achieves even better performance than the "Mono-GAP".

Section 2 overviews the basic architecture of GAP, while Section 3 summarizes each of the component modules. Then, Section 4 describes the parallel and distributed extensions to GAP. Section 5 presents prototype implementations, experiments and evaluations. Section 6 contains some concluding remarks.

2. Basic Architecture

2.1 Design Principles

The bit-string representation of "genotypes" and the simplicity of genetic operations form a good basis for GA-VLSI hardware. Genetic algorithms, in general, apply a sequence of selection, crossover and mutation operations repeatedly to genotype bit-strings within a population. These operations are problem-independent regardless of the bit-string representation of problems, while fitness evaluation of genotypes is problem-dependent.

Consequently, a GA-VLSI system should be composed of two parts: a general-purpose problem-independent part for selection, crossover and mutation operations, and a problem-dependent part for fitness evaluation. The problem-independent part should contain some components, each of which corresponds to each of the operations of selection, crossover and mutation. The problem-dependent fitness evaluation part is designed for a given specific problem.

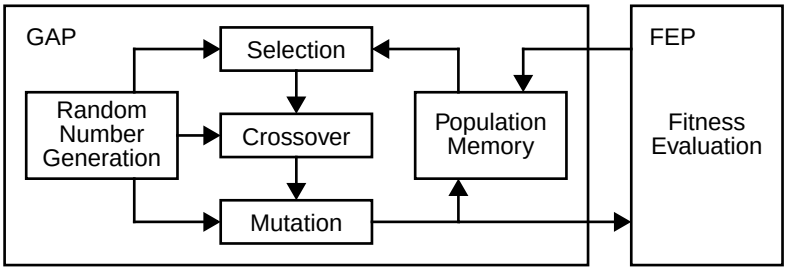

Figure 1. Basic Architecture.

Our design of GA-VLSI follows the above principle as outlined in Figure 1. GAP itself is a problem-independent GA-VLSI, and works in conjunction with a problem-dependent Fitness Evaluation Processor (FEP). GAP contains some modules for random number generation and population control along with modules for selection, crossover and mutation. FEP must be designed for a given specific problem. GAP and FEP are connected via a population memory.

This two-piece design is common to other general-purpose GA-VLSI implementations as well as general-purpose GA software platforms. This is because the basic schemes for selection, crossover and mutation operations are common to most GA applications regardless of bit-string representations of problems. However, some GA applications introduce problem-specific selection, crossover or mutation schemes, and some software platforms provide several schemes for them. Such sophisticated GA applications are out of concern here, and will be addressed in the future.

2.2 Steady-State GA

Many other GA-VLSIs have employed the conventional generational GA. In that scheme, the whole population is updated at once when a generation proceeds. Two sets of population memories are required for the current and next generations respectively, and there is overhead for transferring or switching one memory to the other when a generation proceeds.

Theoretical research on GA has resulted in a new scheme, the "steady-state GA", being proposed [12]. In this scheme, the population is updated continuously; there is actually no concept of generations. New genotypes, as soon as they are created, replace old genotypes with bad fitness values within the population. This scheme has been examined extensively with regard to performance and convergence properties, and is now gradually being adopted by software GA systems.

GAP employs the steady-state GA. Genotypes are passed from GAP to FEP as soon as they are created, and then passed back from FEP to GAP as soon as they are evaluated. Genetic operations and fitness evaluations are overlapped in this way, and the whole system works in a pipeline fashion. Only a single set of population memories is required.

VLSI hardware design for the steady-state GA requires no fundamental revolution from a design for the generational GA. Pipeline processing causes access conflicts to the population memory, however these are resolved by duplication of some registers and delayed updates; the "update" phase of the population memory by new genotypes is delayed until the next "read" phase has been completed.

3. Modules

We must invent algorithms and procedures for the components which are simple enough and yet effective. The best procedure for software implementation may not necessarily be the best procedure for hardware implementation.

3.1 Selection Module

Most other GA-VLSIs, as well as the previous version of GAP, employ the "roulette wheel selection" scheme for genotype selection. The only exception as far as we know is a work by Shackleford et al. [7]. The roulette wheel selection is the most straight-forward scheme: the greater a genotype's fitness is, the more likely that genotype will be selected. Its procedure is as follows:

(1) Sum up the fitness values of all the genotypes within the current population, and generate a random number S_t which is smaller than the sum.

(2) Examine the genotypes in the order in which they appear in the population, and accumulate the fitness values into S_a. When $S_a > S_t$, the genotype under examination is selected.

However, as can easily be seen, this scheme is a bottleneck for VLSI hardware in respect to both circuit size and performance.

In theoretical research on GA, some alternative selection schemes which are better with regard to convergence properties have been studied and exam-

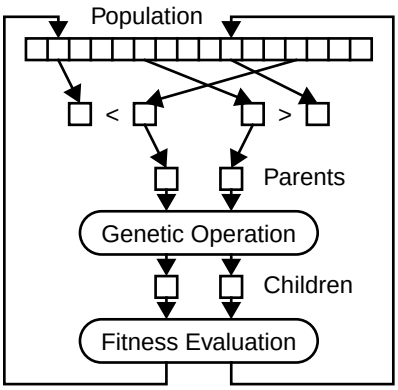

Figure 2. Simplified Tournament Selection.

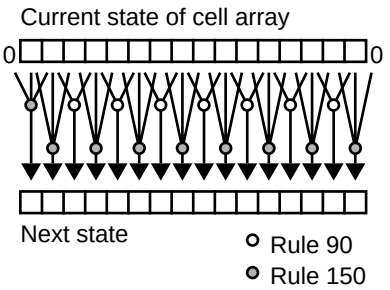

Figure 3. CA-based Random Number Generation.

ined [13]. Investigating such theoretical researches, and following the advice of Anderson [14], we have introduced a selection scheme which is suitable for VLSI hardware. It is named the "simplified tournament selection", since it is a simplified version of the tournament selection scheme. Its procedure, as outlined in Figure 2, is as follows:

(1) Select two genotypes randomly. The better one (with regard to fitness value) is to be "Parent A", with the worse one being "Dead A";

(2) Select two genotypes randomly. The better one is to be "Parent B", with the worse one being "Dead B";

(3) Perform genetic operations on Parents A and B, and create Children A and B;

(4) Substitute Children A and B for Deads A and B, and then discard Deads A and B.

Sato et al. [13] made an extensive study on selection schemes, comparing several schemes including the roulette wheel selection, the tournament selection and more sophisticated ones. They concluded that the roulette wheel selection was the worst, and the tournament selection was among the second best. The best one, which they invented, was too complicated from the VLSI implementation standpoint of view.

We undertook some preliminary examinations for the simplified tournament selection in software prior to designing GAP hardware, and obtained satisfactory results as described later in Section 5. Consequently, we employed the simplified tournament selection scheme for GAP hardware.

3.2 Genetic Operation Modules

The crossover module and the mutation module perform corresponding genetic operations respectively. The schemes for them are the most basic ones: the single-point crossover and the single-point mutation. Both operations are bit manipulations over bit strings of genotypes, therefore they are easy to incorporate into VLSI hardware. Each module takes two random numbers: a probability and an index. When the probability exceeds a pre-defined threshold, the module decodes the index to obtain a point for crossover or mutation, and then performs the operation.

3.3 Random Number Generation Module

This module generates a sequence of pseudo-random bit strings using the theory of linear cellular automata (CA). The CA scheme was proved theoretically to generate better random sequences, in the sense that the sequences had a longer cycle length, than the scheme of linear feedback shift registers (LFSR) which has been widely used [15].

The CA system for random number generation consists of some cells (bits) which update their states according to rules named "90" and "150":

$$\text{Rule 90:} \quad S_i^+ = S_{i-1} \oplus S_{i+1}$$
$$\text{Rule 150:} \quad S_i^+ = S_{i-1} \oplus S_i \oplus S_{i+1}$$

where S_i is the current state of the i-th cell (bit) in the linear cell array (bit string), S_i^+ is the next state for S_i, and $\oplus$ is the "exclusive or" operator. It was proved that a CA system whose cells update their states by the rule list "150-150-90-150-90-...-90-150-90-150" (Figure 3) produces a maximum-length cycle, and has greater randomness than an LFSR system of the same bit length.

From the VLSI implementation standpoint of view, the CA scheme spends more gates than the LFSR scheme. In our experiments, the CA scheme spends 714 gates as shown later in Figure 7, while the LFSR scheme spends 549 for the same bit length. However, the randomness of the random number generation is one of the most crucial in GA implementation, and the CA scheme was proved to be better in this respect. Consequently, we employed the CA scheme.

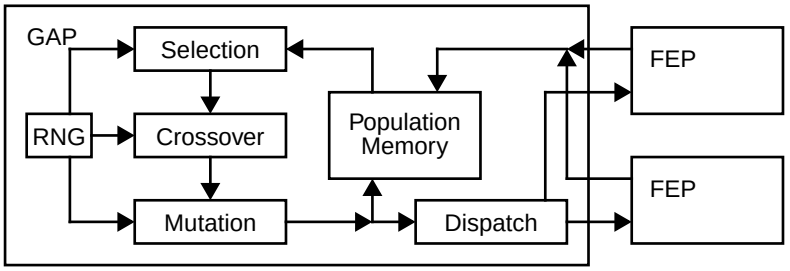

Figure 4. Parallel GA Using GAP.

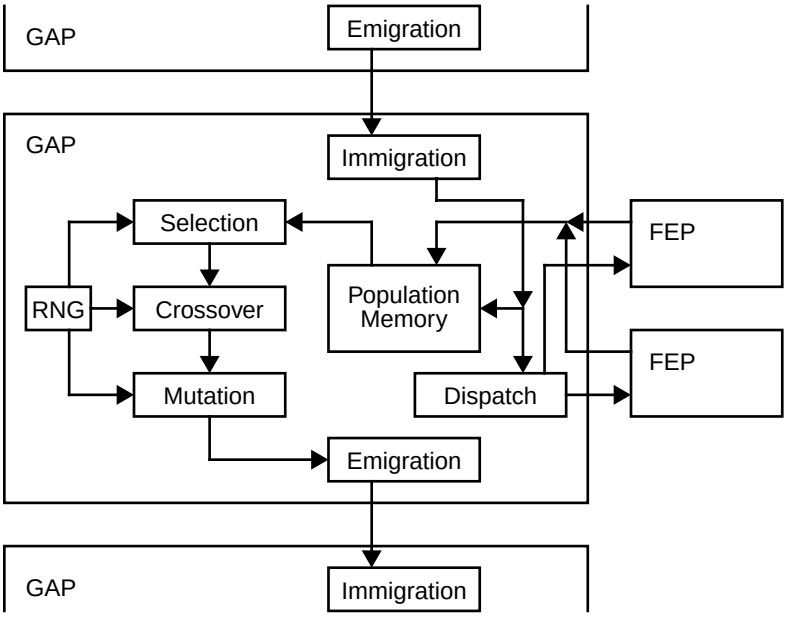

Figure 5. Distributed GA Using GAP.

3.4 Population Memory

As stated earlier, there is a single set of population memories in the system. Each entry in the population memory includes a pair, comprising the bit representation of a genotype and its fitness value.

3.5 Fitness Evaluation Module

This module evaluates fitness for each genotype within the population. Its procedure depends upon the mapping function and the evaluation function for a given problem. The fitness values must be positive and on the-greater-the-better basis. Fitness on the-less-the-better basis in some problems needs to be normalized.

4. Extensions for Parallel and Distributed GA

As problems to be solved become more complicated, FEPs turn into bottlenecks for the overall GA system performance. Therefore, multiplication of FEPs in a system can be effective for improving performance.

In the research area of theories and software for GA, there have already been very many studies on parallel and distributed processing for GA. Parallel GA evaluates fitness values of multiple genotypes simultaneously, and thus realizes fine-grained parallel processing. Distributed GA, or multi-deme-based GA, uses multiple populations which are evolving concurrently, and thus realizes coarse-grained parallel processing.

The basic architecture of GAP hardware design facilitates extensions for parallel GA and distributed GA in line with the studies mentioned above.

4.1 Parallel GA using GAP

The simplified tournament selection scheme

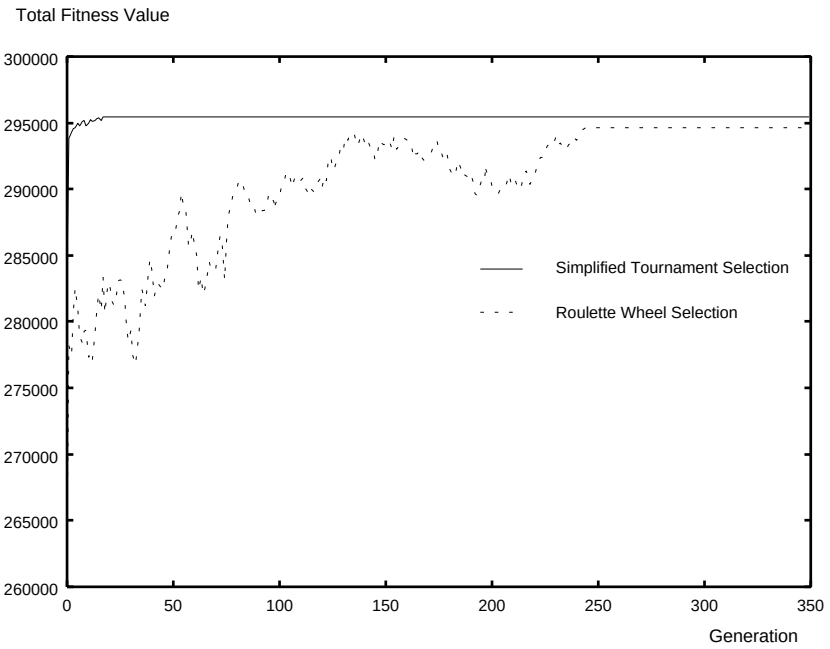

Figure 6. Comparison of Selection Algorithms in Software Implementation.

creates two new genotypes and evaluates their fitness values in every evolution cycle. Therefore, two FEP chips can be connected to a GAP, as shown in Figure 4, and they evaluate the fitness values of the two new genotypes simultaneously. The dispatch module controls the two FEPs. This parallel GA configuration is expected to double the performance of fitness evaluation.

4.2 Distributed GA using GAP

The distributed GA configuration of GAP is composed of multiple GAPs working concurrently. It is important in distributed GA that some of the genotypes should be "migrated" between demes occasionally in order to prevent isolated evolution and premature convergence. Currently, we use the simplest scheme for migration: in every evolution cycle, newly created genotypes are migrated to the next deme.

GAP chips are connected to each other in a ring form, as shown in Figure 5. Newly created genotypes are transferred to the population memory of the next GAP via the emigration and immigration modules. This configuration is expected to accelerate convergence.

5. Implementation and Evaluation

Before attempting implementation in hardware, we programmed both the simplified tournament selection and the roulette wheel selection schemes in software in order to compare their convergence properties. A result for De Jong's function No.1 is shown in Figure 6, in which evolution following the simpli-

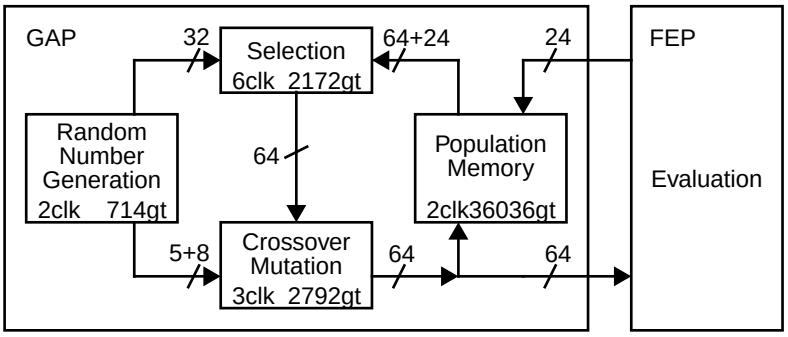

Figure 7. Block Diagram of GAP Prototype.

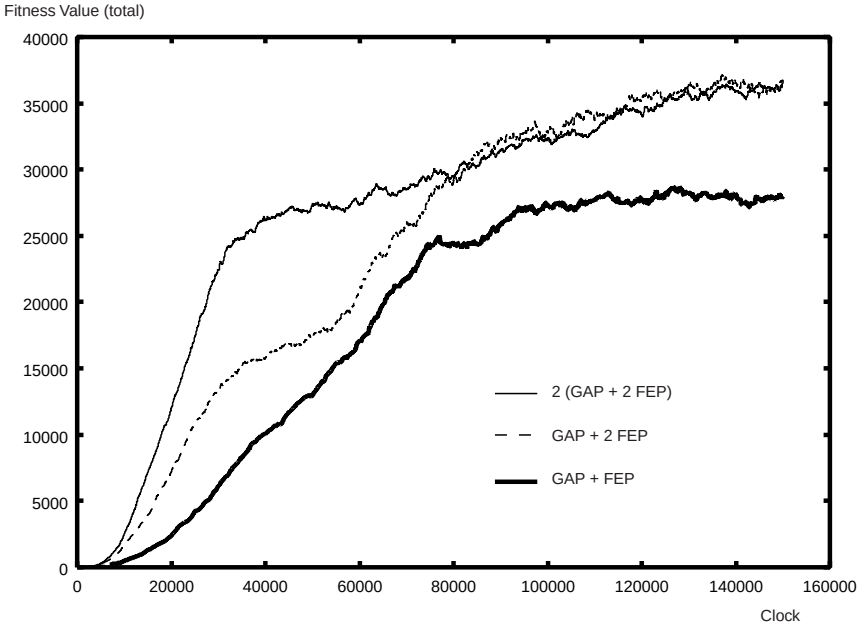

Figure 8. Convergence of Royal-Road Function.

fied tournament selection (the solid line) converges faster up to a higher plateau than evolution following the roulette wheel selection (the dashed line). We confirmed that the simplified tournament selection is, at least, not worse than the roulette wheel selection.

We implemented the VLSI hardware design of GAP using a hardware description language "SFL" [16][17]. The program is approximately 1,000 lines long. We carried out logic simulation and logic synthesis for preliminary evaluation of the design prior to actual VLSI fabrication. We evaluated its complexity, performance and convergence.

The steady-state GA brings a 20% speedup at a 3% increase in circuit size (without regard to the population memory) compared to the generational GA [9]. Reducing the number of population memories from two (in the generational GA) to one (in the steady-state GA) brings reduction of the total circuit size which well compensates the above-mentioned small increase for read/write conflict resolution.

The simplified tournament selection spends 1/6 amount of clock cycles of the roulette wheel selection on average for selection, and the former required 72% of the circuit size of the latter for the selection module [10].

We here present some experiments on parallel and distributed GA using GAP. The specification of the prototype implementations is:

Population size:	256
Genotype bit length:	64
Fitness bit length:	24
Crossover probability:	1
Mutation probability:	1/32

Figure 7 shows the basic block diagram of the prototype. The bit width of the bus is shown along the bus line. Within each module box, the clock cycles (lower left) and the number of gates (lower right) of the module, with the CMOS 0.8μm process technology assumed, are presented for approximate evaluation of the circuit size and performance.

We have three implementations:

(1) GAP with single FEP (basic GA implementation)
(2) GAP with dual FEPs (parallel GA implementation)
(3) Two GAPs, each with dual FEPs (parallel and distributed GA implementation).

The implementation of GAP in (2) spends 1,175 more gates than in (1) for the dispatch module, and the implementation of GAP in (3) spends 1,297 more gates than in (2) for the emigration/immigration modules.

Problem examples used in the experiments consist of:

(1) De Jong's functions Nos. 1 and 2 (very simple and fast to converge): His function suite is widely used for GA system evaluation.

(2) Royal-road function [7][18] (simple but slow to converge): This is also used for evaluation.

(3) Data-partitioning problem [19] (complex and slow to converge): The problem is to generate a set of N–1-dimensional hyperplane boundaries in N-dimensional data space so that every region contains only one data item. Our implementation is dividing

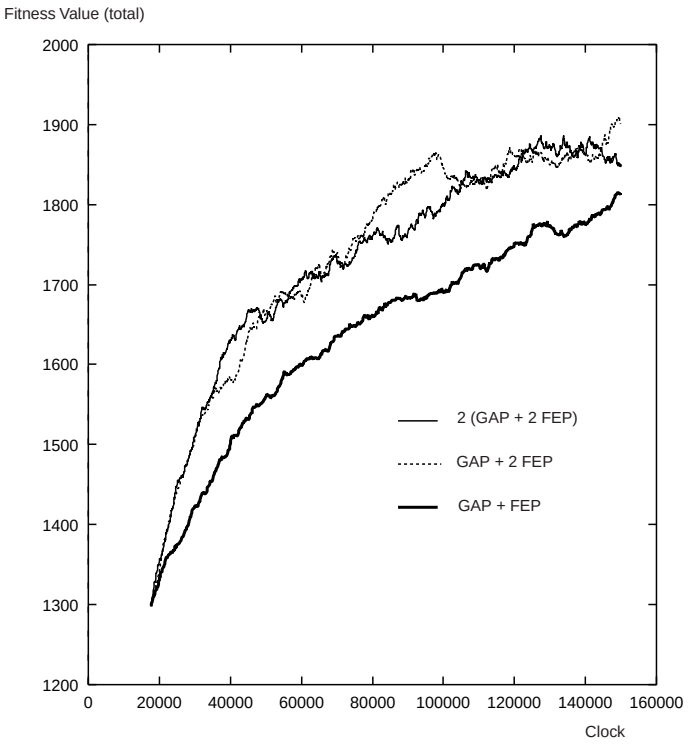

Figure 9. Convergence of Data Partitioning Problem.

a plane with a set of lines. This is one of real-world applications of GA.

Here we show the results of examples (2) and (3). Below is a summary of clock cycles in a single evolution cycle (selection, crossover, mutation and evaluation):

	Royal-road (clk)	Data-partitioning (clk)
GAP + FEP	23	143
GAP + 2 FEP	16	76
2 (GAP + 2 FEP)	17	77

The clock cycles become almost half by the parallel GA configuration, especially in the case of the complex problem of data-partitioning.

Convergence of the royal-road function and convergence of the data-partitioning problem are shown in Figures 8 and 9, respectively. They are both the average of some tens of runs. The Y-axis represents the total fitness value of genotypes in the population; in the distributed GA configuration, it is the average of two populations. As seen in the figures, the parallel and distributed GA configuration exhibits faster convergence than the basic GA implementation, especially in the case of the slow-convergence problem of royal-road.

6. Related Works

Most other studies on VLSI hardware for GA, some notable examples among which are mentioned in Section 1, adopted the roulette wheel selection scheme, which is less suitable to VLSI implementation than our simplified tournament selection scheme as we presented.

The research by Shackleford et al. [7] is most closely related to ours. They proposed their original GA procedure, “Survival-based GA”, which was somewhat similar to the steady-state GA. However, they exhibited the convergence properties of their procedure only empirically. Our research is based on the steady-state GA and a simplified version of the tournament selection. Properties of both the steady-state GA and the tournament selection have already been studied in theories and software implementations for GA.

The idea of parallel GA-VLSIs using more than one fitness evaluation modules was presented in some papers such as [5]. On the other hand, as far as we know, the VLSI design of distributed GA connecting some GA chips using emigration/immigration modules is first presented in this paper. Concrete implementation and simulation evaluation of both the parallel GA-VLSI and the distributed GA-VLSI are also our novelty.

Another interesting topic is “Evolvable Hardware” which is being studied at the Electrotechnical Laboratory in Japan [20]. In that framework, the hardware module itself evolves.

7. Concluding Remarks

This paper presented a VLSI hardware design for GA, which we name GAP (Genetic Algorithm Processor), and its extensions for parallel and distributed GA.

Our prototype implementations proved that the architecture of GAP facilitates two-level parallelization of parallel GA and distributed GA. Our experiments proved that parallel and distributed GA using GAP are effective in performance and convergence improvement. The more complex the problems are, the more eminent the effects is.

We previously studied an adaptive migration scheme for distributed GA in software implementations on a workstation network [21]. The scheme is simple enough for VLSI hardware implementations, and we are now planning to introduce it into GAP. In addition, we are fabricating the design of GAP using the FPGA (Field Programmable Gate Array) technology.

Acknowledgments

We are grateful to Prof. P. G. Anderson (Rochester Institute of Technology) and Prof. M. Yamamura (Tokyo Institute of Technology) for their helpful comments. Comments from anonymous reviewers to the earlier version of this paper are invaluable. We also thank Ms. K. Miller for English proof reading.

References

[1] B. C. H. Turton, T. Arslan and D. H. Horrocks, "A Hardware Architecture for Parallel Genetic Algorithms for Image Registration", *Proc. IEE Colloq. on Genetic Algorithms in Images Processing and Vision*, 11/1-11/6 (1994).

[2] B. C. H. Turton and T. Arslan, "A Parallel Genetic VLSI Architecture for Combinatorial Real-Time Applications – Disk Scheduling", *Conf. Proc. on Genetic Algorithms in Engineering Systems: Innovations and Applications*, 493-500 (1995).

[3] R. Ohshima, N. Matsumoto and K. Hiraki, "Reconfigurable Genetic Algorithm Engine" (in Japanese), *Proc. Third FPGA/PLD Design Conf. and Exhibit*, 541-548 (1995).

[4] Y. Yano, T. Hashiyama and S. Okuma, "On-line Filter Generation for Binary Image Processing Using FPGA", *Proc. 1999 IEEE Int'l Conf. on Systems, Man and Cybernetics*, Vol.5, 565-570 (1999).

[5] S. D. Scott, A. Samal and S. Seth, "HGA: A Hardware-Based Genetic Algorithm", *Proc. 1995 ACM/SIGDA 3rd Int'l Symp. on FPGA*, 53-59 (1995).

[6] I. M. Bland and G. M. Megson, "Implementing a Generic Systolic Array for Genetic Algorithms", *Proc. 1st Online Workshop on Soft Computing* (1996).

[7] B. Shackleford, E. Okushi, M. Yasuda, H. Koizumi, K. Seo, T. Iwamoto and H. Yasuura, "A High-Performance Genetic Algorithm Machine", *Proc. IPSJ Symp. on Information Systems and Technologies for Network Society*, 113-120 (1997).

[8] O. Kitaura, H. Asada, M. Matsuzaki, T. Kawai, H. Ando and Toshio Shimada, "A Custom Computing Machine for Genetic Algorithms without Pipeline Stalls", *Proc. 1999 IEEE Int'l Conf. on Systems, Man and Cybernetics*, Vol.5, 577-584 (1999).

[9] N. Yoshida, T. Moriki and T. Yasuoka, "GAP: Generic VLSI Processor for Genetic Algorithms", *Proc. 2nd Int'l ICSC Symp. on Soft Computing*, 341-345 (1997).

[10] N. Yoshida, T. Yasuoka and T. Moriki, "VLSI Architecture for Steady-State Genetic Algorithms" (in Japanese), *Research Reports on Information Science and Electrical Engineering of Kyushu University*, 3:1, 69-74 (1998).

[11] N. Yoshida and T. Yasuoka, "Multi-GAP: Parallel and Distributed Genetic Algorithms in VLSI", *Proc. 1999 IEEE Int'l Conf. on Systems, Man and Cybernetics*, Vol.5, 571-576 (1999).

[12] L. Davis (ed.), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold (1991).

[13] H. Sato, I. Ono and S. Kobayashi, "A New Generation Alternation Model of Genetic Algorithms and Its Assessment" (in Japanese), *J. JSAI*, 12:5, 734-744 (1997).

[14] P. G. Anderson, Personal communication (1997).

[15] M. Serra, T. Slater, J. C. Muzi and D. M. Miller, "The Analysis of One-Dimensional Linear Cellular Automata and Their Aliasing Properties", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 9:7, 767-788 (1990).

[16] Y. Nakamura, K. Oguri, et al., "High-Level Synthesis Design at NTT Systems Labs", *IEICE Trans. on Inf. & Syst.*, E76-D:9, 1047-1054 (1993).

[17] http://www.kecl.ntt.co.jp/car/parthe/

[18] M. Mitchell and S. Forrest, "Fitness Landscapes: Royal Road Functions", in *Handbook of Evolutionary Computation* (T. Back, D. Fogel and Z. Michalewicz eds.), Oxford (1997).

[19] S. K. Pal, S. Bandyopadhyay and C. A. Murthy, "Genetic Algorithms for Generation of Class Boundaries", *IEEE Trans. on Systems, Man and Cyber.* B28:6, 816-828 (1998).

[20] http://www.etl.go.jp/~ehw/

[21] N. Yoshida and R. Araki, "Efficient Implementation of Distributed Genetic Algorithms on Network of Workstations", *Proc. 2nd Int'l ICSC Symp. on Soft Computing*, 336-340 (1997).