PAPER Special Section on Reconfigurable Systems

Self-Reconfigurable Multi-Layer Neural Networks with Genetic Algorithms

Eiko SUGAWARA[†a)], ***Student Member***, **Masaru FUKUSHI**[††], ***and*** **Susumu HORIGUCHI**[†,††], ***Members***

SUMMARY This paper addresses the issue of reconfiguring multi-layer neural networks implemented in single or multiple VLSI chips. The ability to adaptively reconfigure network configuration for a given application, considering the presence of faulty neurons, is a very valuable feature in a large scale neural network. In addition, it has become necessary to achieve systems that can automatically reconfigure a network and acquire optimal weights without any help from host computers. However, self-reconfigurable architectures for neural networks have not been studied sufficiently. In this paper, we propose an architecture for a self-reconfigurable multi-layer neural network employing both reconfiguration with spare neurons and weight training by GAs. This proposal offers the combined advantages of low hardware overhead for adding spare neurons and fast weight training time. To show the possibility of self-reconfigurable neural networks, the prototype system has been implemented on a field programmable gate array.

key words: *self-reconfiguration, multi-layer neural network, weight training by genetic algorithm, FPGA*

1. Introduction

Neural networks (NNs) are an attractive solution in various applications such as signal and image processing, robotics, and real-time control when no algorithmic approach is available. Those applications often require real-time processing by large scale NNs consisting a number of neurons. Examples of hardware NN systems can be found in [1]–[4]. In today's advanced VLSI technology, large scale NNs can be implemented into single or multiple chips; consequently, high-speed, low-power, large scale NNs can be realized in an extremely small area. Such integrated NNs are expected to be used as embedded systems in practical applications in the real world.

One of the major issues in realizing hardware implemented large scale NNs for real-time processing is a reconfiguration mechanism to change the network configuration. The purpose of reconfiguration is to adapt NNs to a given application, changing the number of neurons and their connections (also referred to as links, alternatively), and acquiring the optimal weights between neurons. In practical use, the presence of faulty neurons should be also considered in the reconfiguration phase, because it is almost impossible for large scale NNs to guarantee all the neurons to be fault-free over the run-time and this condition will generate incorrect calculation results. So far, those two tasks, namely changing network configuration and acquiring the optimal weights, have been studied as different approaches in the area of NNs.

Changing the network configuration of NNs is known as a reconfiguration scheme using redundant elements. The purpose of reconfiguration here is to remove the influence of faulty elements from the NNs, keeping the same number of working neurons. To achieve reconfigurable NNs, some redundant elements, such as neurons and their connections, are incorporated into original NNs in advance so that they can replace faulty elements [5]–[9]. This approach is effective; however, it requires a huge chip area to incorporate redundant elements for large scale NNs.

Acquiring the weights of NNs is widely known as the training problem of NNs. The purpose of the weight training approach is to update each weight by a training algorithm so that the NNs can output optimal/near-optimal responses [10]–[13]. Usually, a back-propagation (BP) algorithm or its modified algorithms are used for the weight training of NNs. The main drawback of this approach is the large computational cost, which is significantly increased for updating a large set of weights. To reduce the computational cost, parallel processing [14], [15] and genetic algorithms (GAs) [16]–[19] are employed for weight training.

When a large scale NN integrated on a chip is used as an embedded system, it is necessary to achieve self-reconfigurable NN that can automatically reconfigure the NN and acquire optimal weights without any help from host computers. Usually, in embedded systems there is not enough space for a host computer and reconfiguration time should be kept small. Murakawa et al. [20] developed a dynamic reconfigurable chip for NNs consisting of a RISC processor and 15 digital signal processors (DSPs). The RISC processor mainly executes the GA to evolve the DSP functions and the interconnection among them to emulate neural processing. However, self-reconfigurable architecture for NNs has not studied sufficiently. To achieve self-reconfigurable NNs, both mechanisms of reconfiguration and weight training need to be implemented and built into the original NNs.

In this paper, we propose an architecture for a self-reconfigurable multi-layer NN employing both reconfiguration with spare neurons and weight training by GAs. A time multiplexing technique and a common bus architecture are adopted in order to significantly reduce both the wiring

Manuscript received December 8, 2003.
Manuscript revised March 4, 2004.

[†]The authors are with the School of Information Science, Japan Advanced Institute of Science and Technology, Ishikawa-ken, 923–1211 Japan.
[††]The authors are with the Graduate School of Information Sciences, Tohoku University, Sendai-shi, 980–8579 Japan.
a) E-mail: esugawa@jaist.ac.jp

area and the number of interconnections. GA is adopted in order to achieve a lower computational cost for weight training. This self-reconfigurable NN offers the combined advantages of low hardware overhead for adding spare neurons and fast weight training time. To show the possibility of self-reconfigurable NNs, the prototype system is implemented on a field programmable gate array (FPGA).

The rest of this paper is organized as follows: Section 2 describes the system architecture for the proposed reconfigurable multi-layer NN using spare neurons and GAs. Section 3 presents a reconfiguration method and the detailed procedures of function shifting and weight training by GAs. Section 4 shows the effectiveness of weight training by GAs in simple pattern recognition problem. Section 5 reports the hardware implementation of the prototype NN and GA training scheme on a commercial FPGA. Finally, Sect. 6 concludes the paper.

2. System Architecture

Throughout this paper, attention is focused on multi-layer NNs since they have wide application areas, in particular, in signal and image processing. For simplicity, we concentrate on a three-layer NN, which is a multi-layer NN with a single middle layer, but our approach can be extended to NNs having more middle layers. Figure 1 shows a three-layer NN example of a multi-layer NN. Each neuron in the middle layer is connected with all neurons of its adjoining layers by weighted links. Let us introduce the following denotations:

L: number of neurons in the input layer,
M: number of neurons in the middle layer,
N: number of neurons in the output layer,

Then we call the three-layer NN, which has L, M, and N neurons in its respective layers, as an L-M-N network and represent it as NN(L, M, N). In this network, the total number of connections is $LM + MN$, and adding one neuron to the middle layer involves $L + N$ additional interconnections. This is why large scale L-M-N networks require a huge chip area for implementing a large number of links.

In order to reduce the number of connections, we use a common bus architecture and a time multiplexing technique. The main idea for reconfigurable multi-layer NNs is a combination of changing network configuration and weight training by GAs. Figure 2 shows the proposed reconfigurable multi-layer NN, which consists of two parts: an NN with spare neurons and a weight training by GAs.

In the NN part in Fig. 2, each layer is connected via a common bus and the output of a neuron is fed to all neurons in the next layer by employing a time multiplexing technique. A selector, placed between each layer, chooses one output of a selected neuron in the previous layer and simultaneously inputs it into all neurons in the next layer. Note that each neuron in the input layer corresponds to an input pin. Neurons in the middle and the output layers have registers to hold weights assigned by the weight controller. A set of weights for each neuron is stored in the weight table.

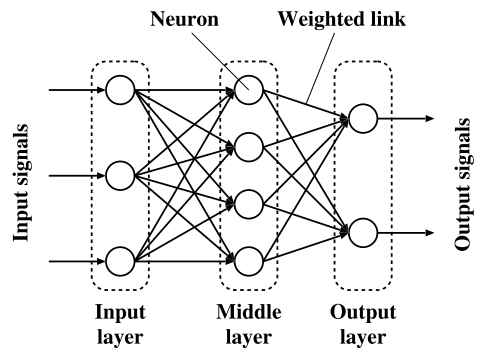


Fig. 1 An example of multi-layer NN with a single middle layer.

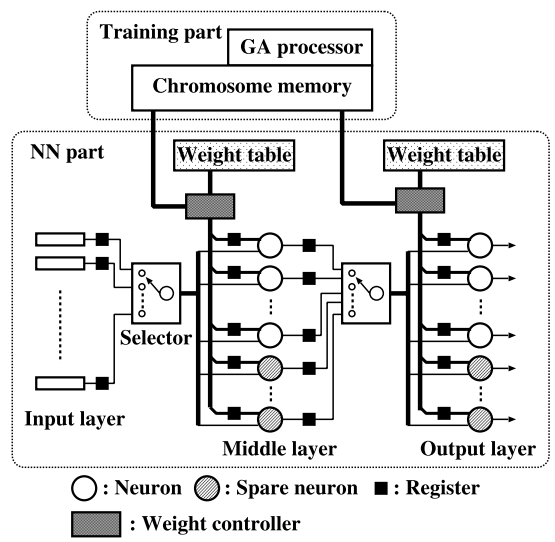


Fig. 2 System architecture of fault tolerant multi-layer NN.

Some spare neurons are added to the middle and the output layer to replace the function of faulty neurons in the same layer. Let s and t be the number of spare neurons in the middle and the output layers, respectively. Then physical network constitution, which include spare neurons, is represented NN(L, M, N) = NN($l, m + s, n + t$), where l, m, and n are the number of neurons required for execution in each layer. The structure of a spare neuron is the same as that of an ordinary neuron. The number of spare neurons in the middle and the output layers can change according to the requirements of specific applications, therefore, it is possible to reconfigure the NN to an arbitrary network size.

In the training part, weights at the middle and the output layers are updated by a GA. In contrast to the well-known BP algorithm, training time of GA is much faster without reducing training accuracy [19]. The training part consists of a GA processor and a chromosome memory. The chromosome memory is a buffer to store chromosomes obtained by GA execution. The best chromosome is stored in the weight table of the NN part and used as the weights for each neuron when the NN part is running.

Details of the faulty neuron replacement procedure and training by GA are described in the next section.

3. Reconfiguration of Neural Networks

3.1 Fault Model

The ability to adaptively reconfigure network configuration for a given application, considering the presence of faulty neurons, is a very valuable feature in a large scale NN. Here, we define our fault mode. In this architecture, all neurons in the middle and the output layers, and the associated parts such as registers to hold weights and wiring to the neuron, are assumed to be faulty. All of those faults can be treated as faults of the associated neuron because their occurrence affects calculation of neuron and results in outputting a wrong value. The faulty neuron that we consider in this paper is a neuron which outputs a value different from the expected one. An expected output is that obtained when there is no fault on neuron, its links, and weights. We assume that except at the input layer, each neuron has a fault detection circuit which can detect faults associated with the neuron. One simple way to realize such circuits may be replication of whole function of a neuron to compare the outputs. However, a detailed discussion of the detection circuit is out of the scope of this paper.

Actually, all neurons, connections, weights, and other NN components are potentially faulty. The common busses, shown by a bold line in Fig. 2, are assumed to be fault-free. This assumption can be justified if those busses are manufactured wide enough. Furthermore, we don't consider faults in weight tables and chromosome memory since an appropriate fault tolerant method for memory arrays can compensate for them. Weight controllers, selectors, and the GA processor are also assumed to be fault-free since their circuit area is quite small and the probability of their being faulty is negligible.

3.2 Reconfiguration Method

In this paper, we define a self-reconfiguration for NNs as (1) changing network configuration to remove faulty neurons, and then (2) obtaining a set of weights for the new network without any help from host computers. In our system, function shifting is used to achieve the first part in the self-reconfiguration scheme and weight training by GA is used to achieve the the second part in that scheme.

Function shifting is a procedure of re-assigning weights for replacing faulty neurons with spare neurons in the same layer. It is used not only to remove the faulty neurons but also to change the number of neurons in the middle and the output layers. Weight training by GA is a well-known technique to obtain weights for NN whose the network configuration is fixed. We employ the same idea to obtain weights for the NN whose faulty neurons are completely removed by function shifting.

By the ability of optimizing weights, a set of weights for correct behavior may be obtained by weight training even when some faults exist. However, employing only weight training is not enough to achieve self-reconfiguration, because there is a kind of fault such that weight training scheme can not cope with. For example, drift fault on a neuron needs hardware compensation [15] because the output of the neuron change randomly. Therefore, we combined weight training by GA with function shifting, and implemented them in hardware together with NN to achieve self-reconfiguration without any help from host computers.

The procedure of our reconfiguration method is as follows:

1. All neurons are classified into faulty and non-faulty neurons by a fault detection circuit.
2. The following processing is performed according to the number of faulty neurons.
   - If the number of spare neurons is greater than or equal to that of faulty neurons, spare neurons are used instead of faulty neurons and weights are re-assigned from the weight table to each neuron. The procedure is completed.
   - If not, faulty neurons are completely removed from the NN and weights for the new network are trained by the GA.

Those procedures are performed whenever the system is initialized or faults are detected. Hence, neurons which were detected as faulty due to the existence of transient faults can be used again if the detection circuits defect no fault at a later time.

This method has an advantage as follows; if the number of faulty neurons is less than that of spare neurons, it is possible for fast reconfiguration without weight training by GA. If not, the network configuration is changed by function shifting and the weights for newly network is obtained by hardware GA.

3.3 Function Shifting

Since spare neurons are added to both layers separately, we describe the replacement procedure for a certain layer, say the middle layer. The same procedure can be applied to the output layer. Several variables are defined to describe the procedure of function shifting, as follows:

Definition 1 All neurons are addressed from the top to the bottom of the middle layer. The address is referred to as the physical ID and denoted as p, where $0 \leq p \leq m + s - 1$.

Definition 2 All fault-free neurons have a logical ID from the top to the bottom of the middle layer. Let $l(p)$ be the logical ID of the p-th neuron, where $0 \leq l(p) \leq m - 1$.

Definition 3 Let $state(p)$ be the state of the p-th neuron of the middle layer, and be defined as follows:

$$state(p) = \begin{cases} 0, & \text{(fault-free)} \\ 1, & \text{(faulty)} \end{cases}$$

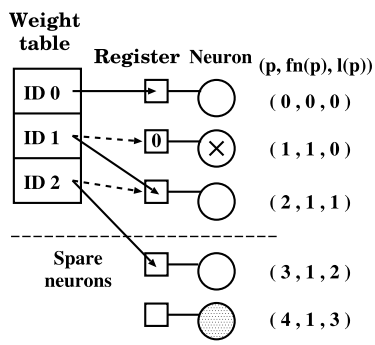


Fig. 3 An example of function shifting. (X: the faulty neuron)

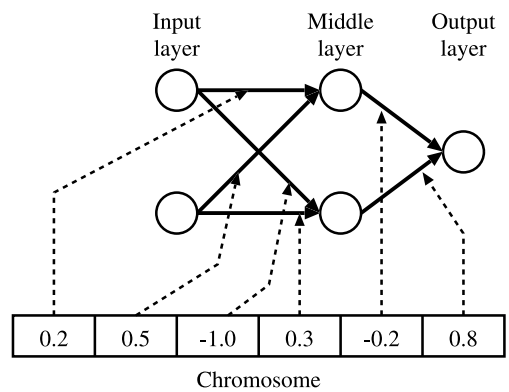


Fig. 4 Chromosome-to-weight mapping.

Definition 4 Let $fn(p)$ be the number of faulty neurons from the top to the p-th neuron and be defined as follows:

$$fn(p) = \sum_{i=0}^{p} state(i)$$

Note that the number of faults is not the total number of faulty neurons which exist in the middle layer.

Assume that all of the above variables are stored in each neuron for function shifting.

The procedure of function shifting is as follows:

1. Initialize logical IDs of all neurons in the middle layer

$$l(p) = p.$$

2. Check the state of neurons and set each $state(p)$.
3. Calculate each $fn(p)$.
4. Set all logical IDs as follows:

$$l(p) = \begin{cases} p - fn(p), & \text{if } state(p) = 0 \\ 0, & \text{otherwise} \end{cases}$$

5. Obtain the corresponding weight with updated logical IDs from weight table if $l(p) < m$.

Figure 3 shows an example of function shifting, when the number of neurons and spares in a layer are 3 and 2, respectively, namely, $m = 3$ and $s = 2$. In this example, the function of faulty neuron (p=1) is shifted to the next neuron (p=2). Our proposed NN can incorporate any number of spare neurons, keeping the increase in chip area for wiring small. Using spare neurons, the function shifting process can achieve a fault-free layer as long as the number of faulty neurons is less than or equal to that of spare neurons.

Note that valid weights are assigned to at most m neurons in step 5 of the function shifting. Thus, our NN can change the system size based on the requirements of specific applications.

3.4 Weight Training by GA

The procedure of weight training by a GA is described here. A chromosome of the GA represents the set of weights of an NN. Each chromosome is the list of real values, each of which maps onto the set of weights of an NN as shown in Fig. 4. Using genetic operators such as selection, crossover, and mutation, the error between the actual output and the desired output is made small. The genetic operators used are roulette wheel selection, uniform crossover, and bit mutation. The probability of crossover and mutation are respectively set to 0.5 and 0.001. The execution of the GA finishes when the end condition is satisfied, that is, the best fitness value becomes 0.01 or the number of generation becomes 10,000. The procedure is as follows:

1. Generate initial chromosomes and evaluate their fitness values.
2. The following processes are repeated until the end conditions are satisfied.
   a. Select two chromosomes from the current population as parents, and reproduce descendants by crossover operation.
   b. Change bits of a chromosome randomly by mutation operation.
   c. Evaluate fitness values of newly generated chromosomes.

To evaluate fitness values, the average error is calculated. Suppose there are n output nodes in an NN. The training set is $\{x^p, t^p | p = 1, \ldots, P\}$, where x^p and t^p is a set of input and desired output for pattern p, respectively, and P is the number of training sets. Let z^p be the actual output when pattern p is input to the network. Then the corresponding error function (E_p) and the average error for all patterns (E_a) are defined as

$$E_p = \sum_{i=1}^{n} (t_i^p - z_i^p)^2. \quad (1)$$

$$E_a = \frac{1}{P} \sum_{p=1}^{P} E_p. \quad (2)$$

When E_a becomes the minimum, the network is considered to be in an optimal state. Then the fitness value of each chromosome is evaluated using Eq. (1) and Eq. (2).

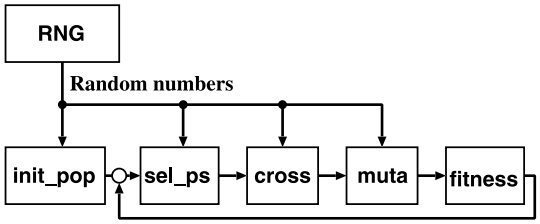


Fig. 5 Block diagram of GA processor.

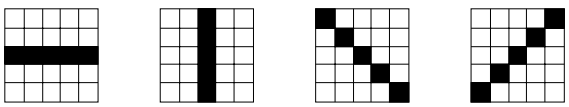


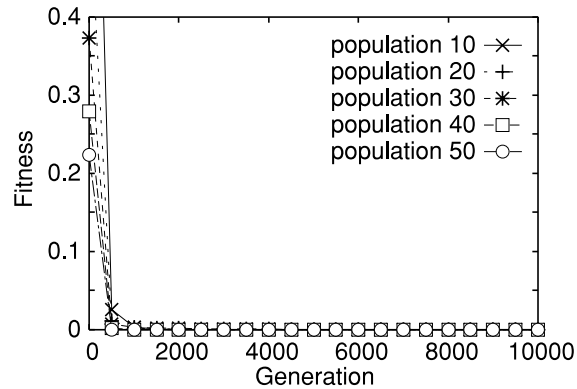
Fig. 6 Training patterns.

The block diagram of the GA processor is shown in Fig. 5. Each unit of the GA processor is executed in pipeline fashion. The `init_pop` in Fig. 5 generates the initial population randomly and stores them in internal memory. The `sel_ps` selects two chromosomes (parents) from the internal memory. The `cross`, `muta`, and `fitness` in Fig. 5 are circuits which execute crossover, mutation, and fitness evaluation, respectively. Random numbers used in the GA operations are generated by a random number generation circuit (`RNG`). The results of the GA execution are stored in chromosome memory, as shown in Fig. 2, and then the best result is used as a set of weights of an NN when it is running.

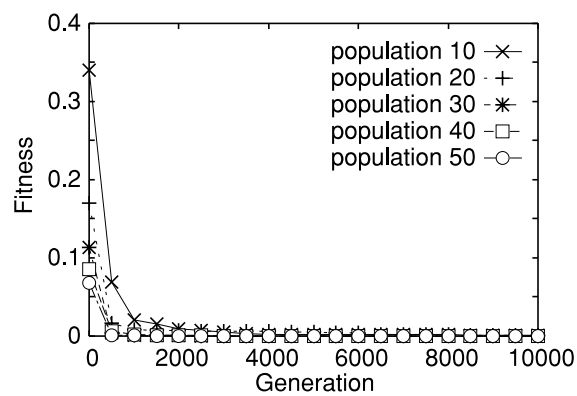
4. Performance Evaluation of Weight Training

In order to investigate the capability of weight training by the GA, we applied it to NN for a simple pattern recognition problem. The goal was to decide the type of a symbol when one symbol was selected from an input pattern set and input to the well trained neural network. Figure 6 illustrates the input pattern set. The size of the input image is 5×5 and each bit is either 0 or 1. The output of the network expresses a specific symbol, that is, one of the four bits corresponding to the recognized symbol becomes 1. First, weights of the 25-15-4 network are trained by the GA. Suppose no spare neuron is now available in the network. The training pattern is the set of four symbols and the desired outputs. Second, pattern recognition is performed by the NN, using the obtained weights, to evaluate the training time and the ratio of correct answers.

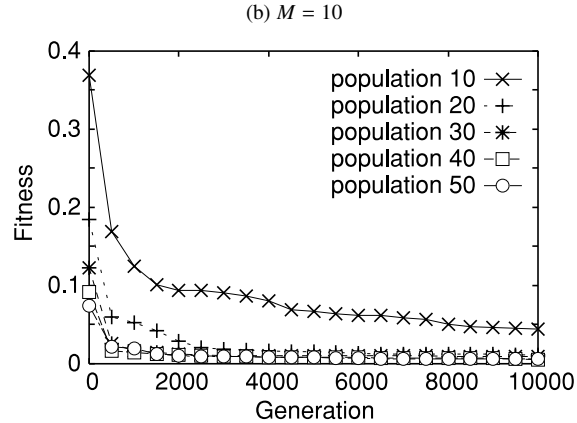
Figures 7 (a)–(d) show the fitness convergence of the best chromosome as a function of the number of generations. The figures illustrate fitness transitions when M = 15, 10, 5, and 2, respectively. All four figures show a common tendency, that is, when population size increases the fitness value is improved, and when M decreases the fitness value also deteriorates gradually. If M is more than 10, fitness values converge at about the 1000th generation and become almost 0. This means that there is almost no difference between the desired output and an actual output. If M decreases to 5, small fitness values can be obtained at



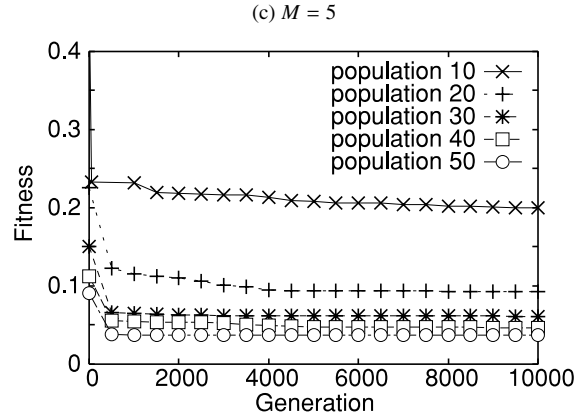
(a) M = 15



(b) M = 10



(c) M = 5



(d) M = 2

Fig. 7 Fitness values over generations. (M: number of neurons in the middle layer)

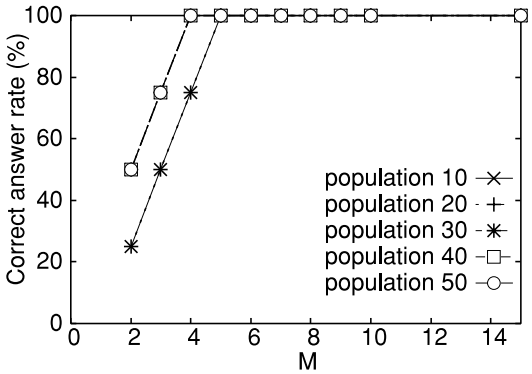


Fig. 8 Correct answer rate as a function of M for 4 symbols.

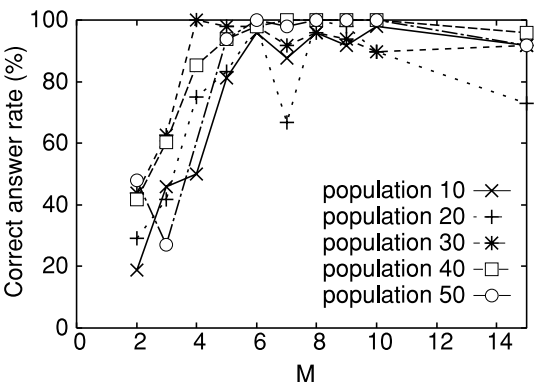


Fig. 9 Correct answer rate as a function of M for 48 symbols.

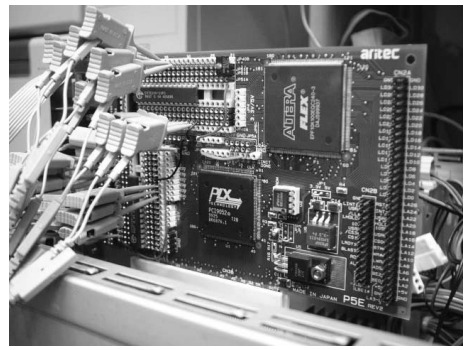


Fig. 10 FPGA board.

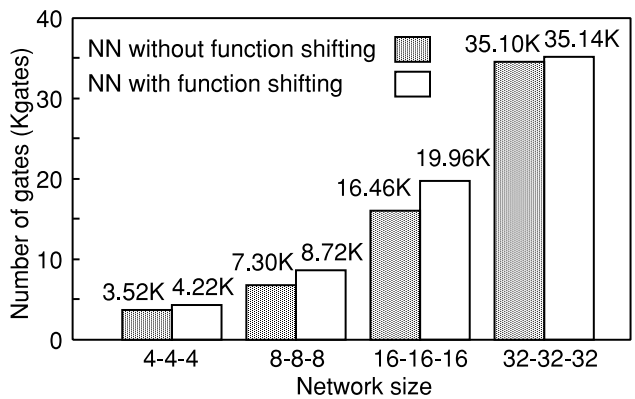


Fig. 11 Hardware cost of NNs.

the 2000th generation when population sizes are 40 and 50. If M decreases to 2, no acceptable fitness is obtained because of the shortage of neurons in the middle layer. Thus, fast training time to obtain small fitness value is achieved by the GA, even if M decreases to 5 because of the occurrence of faults.

Figure 8 shows the correct answer rates for pattern recognition by the NN using the obtained optimal weights. The correct rates are calculated by the same four symbols used as training patterns. When population sizes are 40 and 50, a 100% rate can be achieved even if M is decreased to 4. Figure 9 illustrates the correct rates when the number of input symbols is increased to 48 by adding a random 1 bit error (1 bit reverse) to the original four symbols. Compared to Fig. 8, it is seen that the correct rates are randomly reduced by the influence of errors. Nevertheless, correct rates are over 90% when M is greater than 6 and the population is greater than 40.

5. Hardware Implementation

5.1 FPGA System

To show the possibility of achieving the self-reconfiguration for multi-layer NNs, which is an important feature for large scale NNs in practical industrial applications, the proposed reconfigurable multi-layer NN was designed and implemented on an FPGA. The prototype system was designed in VHDL using the design tool "MAX+Plus II (Altera)" and implemented on the FPGA board "P5E (Alitec)" (Fig. 10), which has an FPGA capacity of about 100 K gates. The hardware costs of the NN part and the training part of Fig. 2 were obtained from the report file of the design tool.

5.2 NN Part

The overall circuit and the behavior of the reconfigurable NN were introduced in Sect. 2. Due to the small chip capacity, each neuron is designed to realize a simple function rather than a complex and high performance function. Here, each weight is 8-bit width and a step function is used as the activation function.

We compare the hardware cost of two NNs: the NN part of the proposed reconfigurable NN and an NN that has no circuit for function shifting but the same network architecture. Figure 11 shows the number of gates used by those NNs, I which the x-axis represents network size and the y-axis represents hardware costs. The number of gates required to implement the 4-4-4, 8-8-8, 16-16-16, and 32-32-32 networks are shown in the figure. In each network, the hardware cost of NN with function shifting shows an increase of about 20% as compared with NN without function shifting. The difference between them corresponds to the hardware cost of the function shifting. To perform the func-

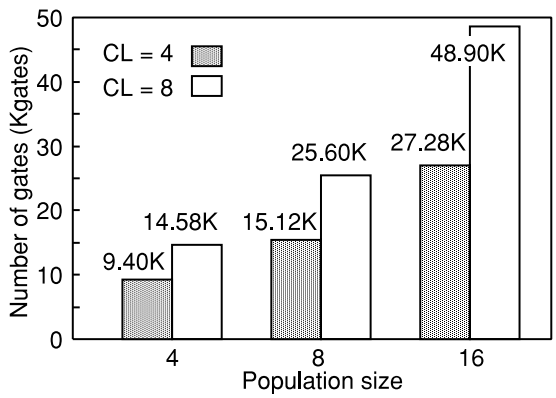


Fig. 12 Hardware cost of training part.

tion shifting, a counter, a weight acquisition circuit, and registers to hold physical ID, logical ID, the number of faulty neurons, etc. were implemented. The hardware overhead of function shifting is small if network size becomes large. Note that the circuits for the function shifting are independent of neuron operations. Therefore, the overhead becomes quite small if each neuron has a complex circuit.

5.3 Training Part

The training part, consisting of the GA processor and chromosome memory, is also implemented in the FPGA. The block diagram of the GA processor is shown in Fig. 5.

Figure 12 shows hardware costs as a function of population size. CL in Fig. 12 means the length of a chromosome, namely, the number of weights. When population size is doubled, the total hardware costs are almost doubled because of the double-sized memory required to store chromosomes. For the same reason, population size does not affect the hardware cost of the GA processor at all, but when CL increases from 4 to 8, the total hardware costs are almost doubled. In this case, the size of some registers in the GA processor is also doubled. However, the number of registers is quite small and total hardware cost of the GA processor is not increased significantly even if CL becomes large. Therefore, the increase of the hardware cost of the training part is heavily dependent on the chromosome memory. Thus, the hardware cost of GA processor does not depend on the scale of the NN.

As a prototype system, a small scale NN was implemented under the limitation of FPGA capacity. Although the NN and training parts were implemented separately, we could confirm their correct behavior through the simulator of the design software and actual signal observations by the logic analyzer.

6. Conclusion

It is very useful for an NN to be able to reconfigure itself and acquire optimal weights automatically without any help of a host computer. In this paper, we proposed an architecture of a reconfigurable multi-layer NN employing both reconfiguration with spare neurons and weight training by GAs. The advantages of the proposed architecture are low hardware overhead for adding spare neurons and fast weight training time. Performance evaluation by the simple pattern recognition shows that the fast and efficient training can be achieved by GAs. The prototype system of the proposed reconfigurable NN was implemented on an FPGA and its correct behavior was confirmed. In future we will implement the proposed architecture on a large FPGA for application to real-world problems.

References

[1] M. Yasunaga, N. Masuda, M. Yagyu, M. Asai, M. Yamada, and A. Masaki, "Design, fabrication and evaluation of a 5-inch wafer scale neural network LSI composed of 576 digital neurons," Proc. IJCNN'90, pp.527–535, San Diego, 1990.

[2] Y. Hirai, T. Ochiai, and M. Yasunaga, "A neural network system composed of 1000 neurons and one million 7-bit synapses," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J84-D-II, no.6, pp.1185–1193, June 2001.

[3] H. Hikawa, "Hardware efficient three-valued multilayer neural network with on-chip learning," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J81-D-II, no.12, pp.2811–2818, Dec. 1998.

[4] T. Kawashima, A. Ishiguro, and S. Okuma, "An architecture of small-scaled neuro-hardware using probabilistically coded pulse neurons," Electrical Engineering in Japan, vol.139, no.4, pp.48–55, 2003.

[5] M.D. Emmerson and R.I. Damper, "Determining and improving the fault tolerance of multilayer perceptions in a pattern-recognition application," IEEE Trans. Neural Netw., vol.14, no.5, pp.788–793, 1993.

[6] D.S. Phatak and I. Koren, "Complete and partial fault-tolerant neural networks," IEEE Trans. Neural Netw., vol.6, no.2, pp.446–456, 1995.

[7] Y. Tohma and Y. Koyanagi, "Fault-tolerant design of neural networks for solving optimization problem," IEEE Trans. Comput., vol.45, no.12, pp.1450–1455, 1996.

[8] C.P. Fuhrman, S. Chutani, and H.J. Nussbaumer, "Hardware/software fault tolerance with multiple task modular redundancy," Proc. Int'l Symp. on Computers and Communications (ISCC), pp.171–177, 1995.

[9] F. Distante, M.G. Sami, R. Stefanelli, and G. Storti Gajani, "Fault tolerant charecteristics of the linear array architecture for WSI implementation of neural nets," Proc. IEEE Int'l Conf. on WSI, pp.113–119, 1991.

[10] C. Khunasaraphan, K. Vanapipat, and C. Lursinsap, "Weight shifting techniques for self-recovery neural networks," IEEE Trans. Neural Netw., vol.5, no.4, pp.651–658, 1994.

[11] Y. Tan and T. Yanya, "Fault-tolerant backpropagation model and its generation ability," Proc. Int'l Joint Conf. Neural Networks, pp.2516–2519, 1993.

[12] D. Simon, "Distributed fault tolerance in optimal interpolative nets," IEEE Trans. Neural Netw., vol.12, no.6, pp.1348–1357, 2001.

[13] N.C. Hammadi, T. Ohmameuda, K. Kaneko, and H. Ito, "Fault tolerant constructive algorithm for feedforward neural networks," Proc. Pacific Rim Int'l Symp. on Fault-Tolerant Systems, pp.215–220, 1997.

[14] U.A. Muller, B. Baumle, P. Kohler, A. Gunzinger, and W. Guggenbuhl, "Achieving supercomputer performance for neural net simulation with an array of digital signal processors," Parallel Computing, vol.14, no.3, pp.329–346, 1998.

[15] K. Yamamori, T. Abe, and S. Horiguchi, "Theoretical performance evaluation of parallel back-propagation algorithms," Proc. Int'l Conf. on PDPTA, pp.1095–1102, 1998.

[16] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.
[17] D. Whitley and T. Hanson, "Optimizing neural network using faster, more accurate genetic search," Proc. Int'l Conf. on Genetic Algorithms (ICGA-89), pp.391–396, 1989.
[18] D. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," Proc. Int'l Joint Conf.on Artificial Intelligence (IJCAI-89), pp.762–767, 1989.
[19] H. Kitano, "Empirical studies on the speed of convergence of neural network training using genetic algorithms," Proc. National Conf. on Artificial Intelligence, vol.2, pp.789–795, 1990.
[20] M. Murakawa, S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, and T. Higuchi, "The GRD Chip: Genetic reconfiguration of DSPs for neural network processing," IEEE Trans. Comput., vol.48, no.6, pp.628–638, 1999.

Eiko Sugawara received the M.S. degree in Information Science from the School of Information Science at JAIST (Japan Advanced Institute of Science and Technology) in 2001. She is currently working towards the Ph.D. degree at Graduate School of Information Science in JAIST. Her research interests are reconfigurable systems and neural network hardware systems.

Masaru Fukushi received the M.S.degree from Hirosaki University in 1997 and a Ph.D in Information Science from the School of Information Science at JAIST (Japan Advanced Institute of Science and Technology) in 2002. He is currently a research associate in the Graduate School of Information Sciences, Tohoku University. His research interests are dependable multi-processor systems, reconfigurable systems and parallel image processing.

Susumu Horiguchi received his M.E and D.E degrees from Tohoku University in 1978 and 1981, respectively. He is currently a full professor in the Graduate School of Information Science, Tohoku University. He was a visiting scientist at the IBM Thomas J. Watson Research Center from 1986 to 1987 and a visiting professor at The Center for Advanced Studies, the University of Southwestern Louisiana and at the Department of Computer Science, Texas A&M University in the summers of 1994 and 1997. He was also a professor in the Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He has been involved in organizing many international workshops, symposia and conferences sponsored by the IEEE, IEICE and IPS. His research interests have been mainly concerned with interconnection networks, parallel computing algorithms, massively parallel processing, parallel computer architectures, VLSI/WSI architectures, and Multi-Media Integral Systems. Prof. Horiguchi is a senior member of the IEEE Computer Society, and a member of the IPS, and IASTED.