Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration

James G. Eldredge and Brad L. Hutchings
Dept. of Electrical and Computer Eng.
Brigham Young University
Provo, UT 84602
Tel: (801) 378-2667
Email: hutch@ee.byu.edu

Abstract

Run-time reconfiguration is a way of more fully exploiting the flexibility of reconfigurable FPGAs. The Run-Time Reconfiguration Artificial Neural Network (RRANN) uses run-time reconfiguration to increase the hardware density of FPGAs. The RRANN architecture also allows large amounts of parallelism to be used and is very scalable. RRANN divides the backpropagation algorithm into three sequentially executed stages and configures the FPGAs to execute only one stage at a time. The FPGAs are reconfigured as part of normal execution in order to change stages.

Using reconfigurability in this way increases the number of hardware neurons a single Xilinx XC3090 can implement by 500%. Performance is effected by reconfiguration overhead, but this overhead becomes insignificant in large networks. This overhead is made even more insignificant with improved configuration methods. Run-time reconfiguration is a flexible realization of the time/space trade-off. The RRANN architecture has been designed and built using commercially available hardware, and its performance has been measured.

1 Introduction

In many cases, reconfigurable Field Programmable Gate Arrays (FPGAs) are used as an easy way to prototype digital logic circuits and as a way to achieve cheap, low-volume, custom VLSI. The ease with which a designer can develop a circuit and then configure an FPGA to implement that circuit is the primary reason. This flexibility comes at the cost of decreased performance due to the overhead involved in making a device fully programmable. Because of this overhead, an FPGA provides less functionality per unit area of silicon and longer communication delay times than gate-array or full-custom implementations [1]. In a conventional *static* design (one where the FPGA's configuration does not change) this flexibility is wasted, while its overhead remains.

The efficient use of FPGAs beyond prototyping and low-volume production requires that the FPGA's flexibility be used directly. One way of exploiting this flexibility is through run-time reconfiguration. Run-time reconfiguration divides an algorithm into sequentially executed stages; at run-time, only one stage is configured on the FPGAs at a time. When one stage completes, the FPGAs are reconfigured with the next stage. This process of "configure and execute" is repeated until the algorithm has completed its task. Because only one stage of the algorithm is actually using hardware at any given time, there are more hardware resources available for use by each stage. These additional hardware resources can be used to improve the performance of the active stage.

This paper describes the Run-Time Reconfiguration Artificial Neural Network (RRANN). RRANN uses run-time reconfiguration to increase the number of hardware neurons implemented on a single Xilinx XC3090. It does this by dividing the backpropagation algorithm into three stages, feed-forward, backpropagation, and update and configuring the FPGAs for only one stage at a time. Because of run-time reconfiguration, RRANN was able to increase the number of available hardware neurons by 500% per FPGA.

As the RRANN architecture uses the backpropagation algorithm for learning, we begin by describing the basics of the backpropagation algorithm for neural networks. Then, the RRANN architecture is presented and explained in some detail. Next, the performance of the RRANN architecture is given, and then some conclusions are made about the project.

2 The Backpropagation Algorithm

The backpropagation learning algorithm [2] provides a way to train a multilayered feed-forward neural network such as the one shown in Figure 1.

It begins by feeding input values forward through the network according to (1) and (2).

$$net_i = \sum_{j \epsilon A} O_j W_{ji} \quad \forall\, i:\ i \,\epsilon\, B, \tag{1}$$

where A is the set of all neurons in a layer, B is the set of all neurons in the next layer, O_j is the output (or activation) for neuron j, and W_{ji} is the weight assigned to the connection between neurons j and i. Equation (1) takes the output values (or activations) from a layer and feeds them forward to the next layer through weights. This operation is performed for each neuron in the next layer producing a *net* value for it. The *net* value is the weighted sum of all the activations of neurons in the previous layer, and each neuron's *net* value is then applied to (2), known as the activation

0-8186-5490-2/94 $03.00 © 1994 IEEE

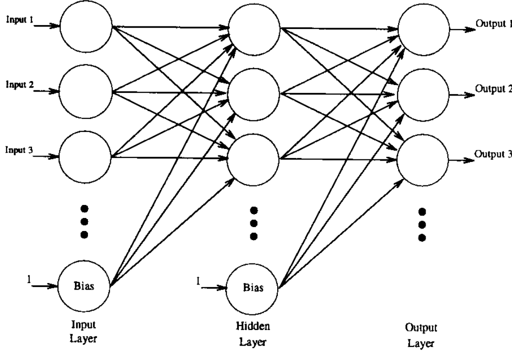

Figure 1: A Multilayer Feed-Forward Neural Network.

function, to produce that neuron's activation.

$$O_i = f(net_i) = \frac{1}{1 + e^{-net_i}}. \tag{2}$$

After all neurons in the network have an activation value associated with them for a given input pattern, the backpropagation algorithm continues by finding errors for every non-input neuron. This process begins by finding the error for the neurons on the output layer according to (3).

$$\delta_i = f'(net_i)(T_i - O_i) \quad \forall\, i:\ i \in C, \tag{3}$$

where C is the set of all output neurons, T_i is the target output for neuron i, $f'()$ is the first derivative of (2), and δ_i is the error for neuron i. The errors found for the output neurons are then propagated back to the previous layer so that errors can be assigned to neurons contained in hidden layers. This calculation is governed by (4).

$$\delta_i = f'(net_i) \sum_{j \in E} \delta_j W_{ij} \quad \forall\, i:\ i \in D, \tag{4}$$

where D is the set of all neurons in a non-input layer and E is the set of all neurons in the next layer. This calculation is repeated for every hidden layer in the network.

After every non-input neuron has an activation and error associated with it, the network's weights can be updated. First, the amount by which each weight should be changed is found by calculating (5).

$$\Delta W_{ij} = C_l O_i \delta_j \quad \forall\, i, j:\ i \in A,\ j \in B, \tag{5}$$

where C_l, known as learning rate, is a constant controlling the size of weight changes, and ΔW_{ij} is the weight change between neuron i and j. The weight is changed by evaluating (6).

$$W_{ij_{t+1}} = W_{ij_t} + \Delta W_{ij}. \tag{6}$$

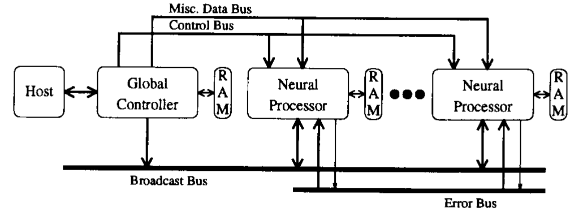

Figure 2: The General RRANN Architecture.

3 The RRANN Architecture

The RRANN architecture divides the backpropagation algorithm into the sequential execution of three stages known as feed-forward, backpropagation, and update. The feed-forward stage is responsible for taking input patterns and propagating them through the network assigning an activation (output) to every neuron according to Equations (1) and (2). The backpropagation stage (expressed by Equations (3) and (4)) finds the output errors and propagates them backward through the network in order to find errors for neurons contained in hidden layers. After every non-input neuron has been assigned an error value, the update stage (Equations (5) and (6)) begins operation. The update stage uses activation and error values found by the previous two stages to compute the amount by which weights should be changed and updates all weights with these changes. The completion of the update stage marks the end of the evaluation of one training pattern. This process is then repeated for all training patterns until the network is sufficiently trained.

RRANN contains a global controller occupying one FPGA and many neural processors occupying the balance of the available FPGAs (Figure 2). The global controller is mainly responsible for sequencing the execution of local hardware subroutines on the neural processors and for supplying key data to the neural processors (such as input patterns). Each neural processor contains six hardware neurons and local hardware subroutines implemented with state machines. The neural processors are responsible for performing all computations needed for the backpropagation algorithm. Each neural processor also has a local RAM associated with it. This RAM is used to store weight and target output data and as a scratch pad to buffer *net* and error values.

In order to use hardware efficiently, RRANN utilizes bit-serial hardware to do mathematical computations. Also, as with other projects [3, 4], RRANN uses look-up tables to implement the activation function (Equation (2)) and its derivative.

3.1 The Feed-Forward Stage

The feed-forward stage computes Equations (1) and (2), and the feed-forward neural processor, shown in Figure 3, contains the hardware for these computations. The feed-forward neural processor contains six hardware neurons that compute the activation-weight $(O \cdot W)$ products and accumulate these products. The

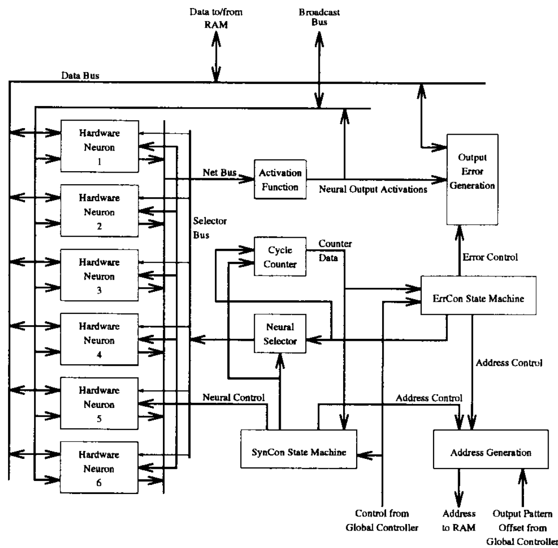

Figure 3: The Feed-Forward Neural Processor.

neural selector decides which hardware neuron and neural processor is currently supplying activation data to the broadcast bus. The block labeled *activation function* in the figure produces Equation (2). It does this by using a *net* value supplied by a hardware neuron as an index into a look-up table. The cycle counter is used by the state machines to sequence events, and the address generation unit produces the addresses needed to access the local RAM. The output error generation unit finds the difference between target outputs and the network's outputs. This is technically part of the backpropagation stage (Equation (3)), but it is done here for convenience. Finally, the SynCon and ErrCon state machines implement subroutines that control the evaluation of the network equations.

To begin the feed-forward stage, the global controller broadcasts an input value to the neural processors. This broadcast is done over a time-multiplexed bus one input value at a time [4]. As each input value is placed on the bus, the neural processors read it and do the appropriate weight multiplication for neurons in the first hidden layer. After the multiplication is complete, the product is accumulated with those from previous input iterations (Equation (1)), and the next input is broadcast over the bus. This method of interconnection allows for a large amount of parallelism to occur as all hardware neurons perform a multiply concurrently. It also allows for a large amount of scalability because hardware neurons can be added to the network by simply connecting them to the bus. Also, each hardware neuron only requires one multiplier; this allows for the development of one generic neuron that can be used with networks of arbitrary size.

After the evaluation of the first hidden layer is completed, the neural processors become responsible for broadcasting activation values to be used as inputs to the next layer. Before the evaluation of another layer begins, the hardware neurons buffer the *net* values calculated for the first hidden layer. Then, each hardware neuron in turn evaluates (2) for its buffered *net* value and broadcasts the result over the broadcast bus to all other hardware neurons including itself. This process iterates as the hardware neurons evaluate (1) and buffer the new *net* values as was done for the first hidden layer.

If the network has four layers, the above process is again repeated for the output layer. After the *net* values for the output layer have been found, they are used to evaluate (2), and the results are compared with the input pattern's target outputs (T_i). These compare operations ($T_i - O_i$) are computed in parallel, and the differences are buffered for use by the backpropagation stage.

3.2 The Backpropagation Stage

The backpropagation stage computes Equations (3) and (4), and the backpropagation neural processor, shown in Figure 4, contains the hardware to do these computations. The hardware neurons find the error-weight products ($\delta \cdot W$) during the evaluation of (4) and also buffer error values. The input and output sum blocks compute the summation of (4). The output register provides an index (a *net* value) into the activation function derivative look-up table, and the activation function derivative look-up table supplies f' to the multiplier. This multiplier finds ($f' \cdot \sum$) to complete the computation of (4). This final product is check for overflow and underflow by the overflow/underflow block. The neural selector determines the hardware neuron and neural processor Equation (4) is being computed for. The cycle counter aids the state machines in sequencing operations, and the address generation unit supplies addresses to the local RAM. Two multiplexors are also used to direct data to their proper locations. The GetErr, OutErr, and ErrComp state machines control this hardware to correctly calculate (3) and (4).

The backpropagation stage begins by finding the errors for the output layer using the differences found during the feed-forward stage according to (3). This operation can be executed with a large degree of parallelism, is purely local, and, therefore, does not require any communication of data between hardware neurons.

Erdogan and Hong [5] point out a stumbling block to parallelism in the calculation of (4). This problem occurs when error values are broadcast back through the network in a similar manner as activation values during the feed-forward stage. The weight values needed for the multiplication after the broadcast are not locally available to the hardware neuron that needs it. Erdogan and Hong suggest weight value duplication as a possible solution. RRANN solves the problem in a fundamentally different way that avoids weight duplication and error broadcasting.

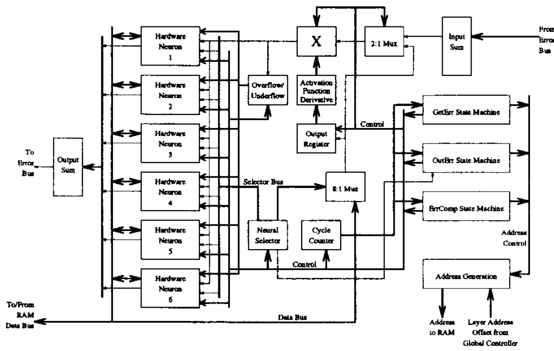

Figure 4: The Backpropagation Neural Processor.

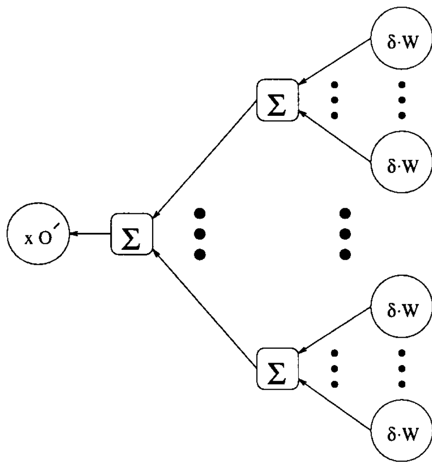

Figure 5: Computing Errors for Hidden Neurons.

RRANN completes the computation of Equation (4) for each hidden neuron in the network (including the summation) before beginning (4) for another hidden neuron. This operation is illustrated by Figure 5. It begins as each output neuron calculates the error-weight product ($\delta \cdot W$) in parallel. Each neural processor then sums the products generated by its hardware neurons, and this partial sum is then communicated to the targeted neural processor. This target neural processor receives these partial sums and completes (4) by adding the partial sums together and then multiplying by the appropriate activation derivative.

Computing Equation (4) in this manner achieves the same level of parallel execution as the feed-forward and update stages. This is an important advantage that overcomes the problem of non-local weight distribution discussed by Erdogan and Hong [5] without requiring weight duplication or broadcasting. However, a small amount of scalability is lost as each neu-

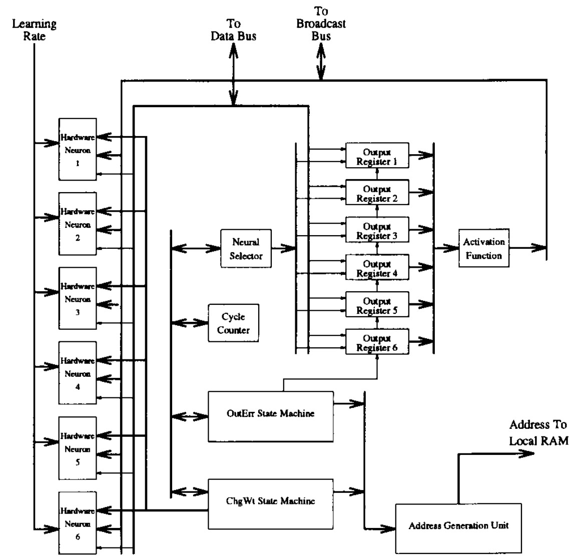

Figure 6: The Update Neural Processor.

ral processor requires its own line of communication. As the number of neural processors grow, the number of error communication lines grow. This increase in communication lines also necessitates an increase in adder circuitry for merging the partial sums. These requirements are minimized, however, by the use of bit-serial data representations. The communication lines are only one bit wide, and bit-serial adders only require one Xilinx Combinational Logic Block (CLB).

3.3 The Update Stage

The update stage computes Equations (5) and (6), and the update neural processor, shown in Figure 6, contains the hardware needed for these calculations. The update hardware neurons compute (5) using error (δ) values retrieved from local RAM, activation (O) values supplied over the broadcast bus, and learning rate (C_l) supplied by the global controller. After finding the ΔWs, the update hardware neurons immediately use these data to update the corresponding weights. The output registers hold *net* data retrieved from local RAM and supply these data as indexes into the activation function look-up table. The neural selector decides which neural processor supplies activation data to the broadcast bus and which output register supplies input to the look-up table. The cycle counter supplies sequencing information to the state machines, and the address generation unit produces addresses for the local RAM. The OutErr and ChgWt state machines control the above hardware to achieve the proper execution of (5) and (6).

The update stage begins in a similar manner to the feed-forward stage with the global controller broadcasting input values to the first hidden layer. The hardware neurons then use the activation and error

values found by the first two stages to compute (5). The ΔW produced is immediately used in the calculation of (6), and the new weight value is copied over the old value. This process is repeated for every weight in the hidden layer in a parallel fashion as each input presents its value. Then RRANN begins changing the weights between the first hidden layer and the output (in a three layer network) or the second hidden layer (in a four layer network). This is again repeated, in the case of a four layer network, for the weights between the second hidden layer and the output layer.

The completion of the update stage marks the end of the evaluation of one complete training pattern. The three stages are repeated for each pattern in the training set (*one epoch*), and new epochs are run until the network is sufficiently trained.

3.4 Run-Time Reconfiguration Between Stages

As was just explained, RRANN divides the backpropagation algorithm into three sequentially executed stages, and then configures only one of the stages on the FPGAs at a time. This process is shown in Figure 7 for one *pass* through all three stages.

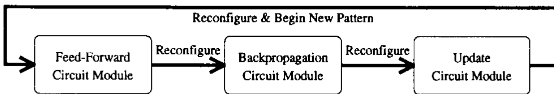

Figure 7: RRANN's Run-Time Reconfiguration.

A separate circuit module was developed for each stage of execution. Each circuit module followed the general architecture shown in Figure 2 consisting of a global controller and many nearly identical neural processors. As one circuit module finished (indicating the completion of the corresponding stage), all FPGA hardware was reconfigured with the next stage's circuit module.

4 Implementation and Evaluation

The RRANN architecture has been built and tested using a commercially-available FPGA board (the X12 board, previously referred to as the SCA board) produced by National Technology Inc. The X12 board is an ISA compatible expansion board that can be used in a host IBM-compatible PC, and it can be populated with up to 12 Xilinx XC3000 series FPGAs. The host PC stores all configuration information for the FPGAs, monitors the progress of each stage of execution, and supplies the appropriate configuration data to the X12 board.

With the X12 board populated with Xilinx XC3090s, reconfiguration time was measured at approximately 30 ms. Compare this to the minimum reconfiguration time for an XC3090 of approximately 7 ms. The reason for the X12 board's longer reconfiguration time is due to the host's low bus bandwidth. In a newer version of the X12 board, this problem is solved through bus mastering and configuration data caching. Therefore, the following analysis will assume the 7 ms reconfiguration time.

	Feed-Forward	Backpropagation	Update
Neural Processor	260	253	224
Global Controller	97	62	98

Table 1: The CLB Requirements of Circuit Modules.

All circuitry was designed and simulated using Mentor Graphics' schematic capture software, and these circuits were implemented on FPGAs using Xilinx's XACT software. Using run-time reconfiguration, six hardware neurons were implemented on each Xilinx XC3090 FPGA. The RRANN circuitry successfully simulated at 10 MHz and operated at 14 MHz. With improved pipelining, especially between the neural processor and local RAM, it is estimated that the circuitry could operate in excess of 40 MHz. Part utilization (in Xilinx XC3000 CLBs) for the feed-forward, backpropagation, and update stages are summarized in Table 1. The RRANN circuitry was tested on neural networks requiring four XC3090s. For these networks, weight convergence was observed, and the networks made useful generalizations about the training patterns presented them [6].

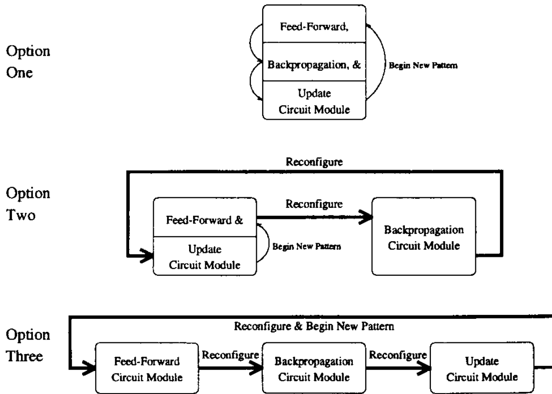

Figure 8: Other Run-Time Reconfiguration Options.

For comparison, two other options for implementing the RRANN architecture have been investigated. The first option combines all three stages of execution into the same circuit module, and this circuit module is then configured onto the FPGAs. Reconfiguration is not required for this option as all stages are present in the one configuration. The second option combines the feed-forward and update stages into one circuit module and the backpropagation stage into another. Reconfiguration is needed after the completion of the feed-forward and backpropagation stages but not after the update stage. These two options have been studied theoretically, and no hardware has been developed for

them. The design explained in the previous section is referred to as option three in the proceeding text. Figure 8 shows these additional two options compared with option three (Figure 7). Again, this third option was implemented and tested using the X12 board and XC3090 FPGAs.

Run-time reconfiguration improves hardware utilization allowing more hardware neurons to be implemented per FPGA. It is estimated that one hardware neuron per FPGA is possible for option one, and four hardware neurons per FPGA for option two. Six hardware neurons per FPGA were attained for option three, yielding a 500% increase in hardware neuron densities over option one. This increase in hardware neuron density is made possible because inactive circuitry is not using FPGA resources. As an example of this, consider the input sum, output sum, and activation derivative blocks of the backpropagation stage (Figure 4). The feed-forward and update stages do not need this circuitry, and, through run-time reconfiguration, they do not implement it. The FPGA resources freed in this example are then used to implement more hardware neurons.

The cost of this improved utilization is the time it takes to reconfigure the FPGAs with a different circuit module. As stated previously, option one requires no FPGA reconfigurations adding 0 ms to overall execution time. Option two requires two reconfigurations adding 14 ms per pass to execution time. Option three requires three reconfigurations adding 21 ms per pass to execution time. Execution time t_E for one pass through all three stages of the RRANN architecture is given by

$$t_E = t_C + t_R, \tag{7}$$

where

$$t_C = \frac{(n \times 148 + (l-1) \times 13 + 282)}{f_C}$$

is the time spent doing computations, n is the number of non-output neurons, l is the number of layers, f_C is the clock frequency (14 MHz), and t_R is the reconfiguration time for the given option. The reconfiguration time t_R remains constant for each option regardless of the network's size. However, computation time t_C grows linearly with n and, therefore, with the number of FPGAs. For option one, t_E is dominated only by t_C because $t_R = 0$ ms. However, t_E for options two and three is dominated by t_R for small numbers of FPGAs. As the number of FPGAs is increased, t_C begins being more significant to t_E. The reconfiguration overhead is, therefore, more dominant for smaller networks.

The Performance of the RRANN architecture for all three options is shown in Figures 9 and 10. Figure 9 shows how the total computational power of the network, measured in weight updates per second (WUPS), grows as a function of XC3090s used. Options two and three are clearly dominated by reconfiguration overhead for small numbers of FPGAs. For larger numbers (> 23) of FPGAs, computational time begins to be more dominant, and run-time reconfiguration begins to pay off with better performance.

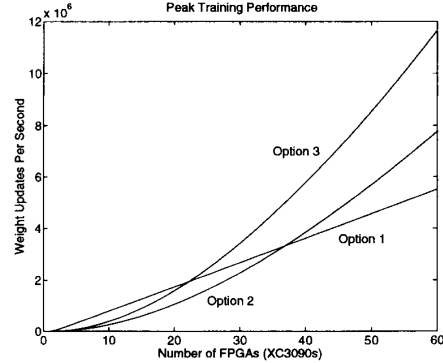

Figure 9: RRANN's Performance (14 MHz).

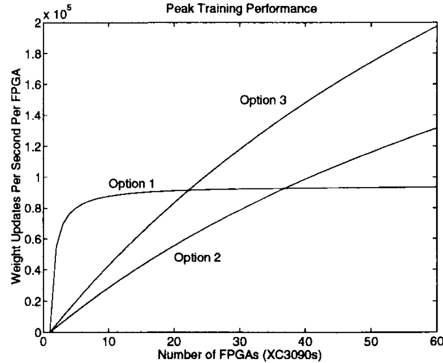

Figure 10: RRANN's Performance per XC3090 (14 MHz).

Figure 10 shows the computational power that each XC3090 is providing as a function of the total number of XC3090s used. Option one reaches peak output per FPGA of 94.6×10^3 WUPS relatively quickly. Again because of reconfiguration overhead, options two and three require larger networks before higher performance is possible. The performance of each FPGA continues to grow asymptotically for these two options until reaching 378×10^3 WUPS for option two and 568×10^3 WUPS for option three.

After the backpropagation algorithm completes operation and the neural network has been sufficiently *trained*, the network is then used to compute outputs for new input patterns. This is known as *operational mode*, and it requires the computation of Equations (1) and (2) only. These equations are totally implemented by the feed-forward stage, and thus, in operational mode, the backpropagation and update stages can be totally discarded. Also, because only one stage is used, no FPGA reconfigurations are needed. This again emphasizes run-time reconfiguration's main advantage: Only that hardware presently needed uses

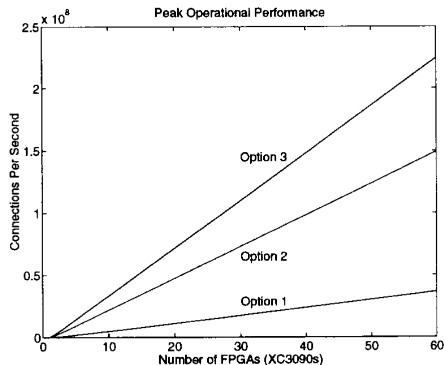

Figure 11: RRANN's Operational Performance (14 MHz).

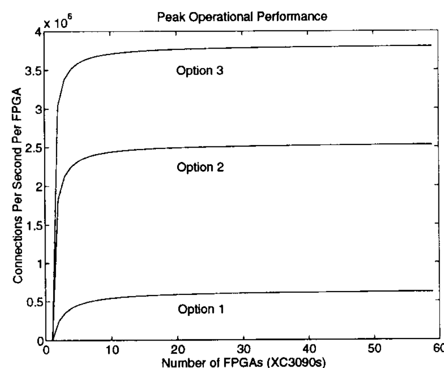

Figure 12: RRANN's Operational Performance per XC3090 (14 MHz).

valuable FPGA resources.

The operational mode performance is shown in Figures 11 and 12. This performance is measured in connections per second (CPS), and the figures show that option three is clearly better. This result is not surprising because only the feed-forward stage is needed, and reconfiguration overhead is eliminated. Also, option three implements more neural processors per XC3090 allowing it to do more neural computations.

In order to make run-time reconfiguration more feasible, the significance of reconfiguration overhead needs to be reduced. One way of doing this is by making computation time more significant at a faster rate. As previously noted, RRANN's implementation of the backpropagation algorithm causes computational time t_C to grow linearly with the number of hardware neurons and, hence, FPGAs. If, for a different algorithm, this rate of growth were $n \log n$ or n^2, t_C would dominate t_R much sooner.

Another way of decreasing the significance of reconfiguration overhead is by finding faster methods for reconfiguring FPGAs (i.e. reducing t_R directly). Using Xilinx parts as an example, the internal data path used to load configuration data could be widened from one to eight bits, and the maximum configuration clock rate could be increased from 10 MHz to perhaps 50 MHz. Negligible reconfiguration times could be realized through an FPGA that is capable of storing multiple configurations. Configuration data for several configurations would be loaded and stored on the FPGA at the beginning of execution. The active configuration could be selected by a few dedicated external pins, and, after its execution is complete, another configuration could be activated by manipulating the select lines and waiting a few microseconds. Of course, internal configuration memory is not free, and an FPGA containing enough memory to hold four configurations might otherwise be able to have four times as many CLBs instead. This is the case with any configuration improvement; the improvement probably increases the part's programmability overhead and reduces the part's functionality. Issues such as these should be studied in order to find out what improvements can be made and the amount of additional overhead they would create.

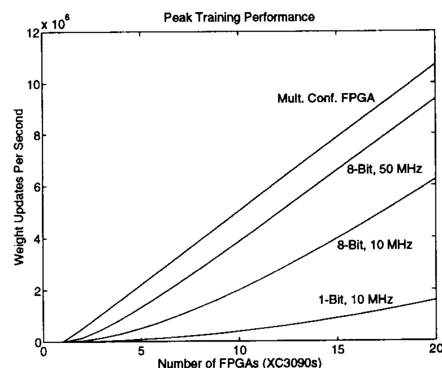

Figure 13: Effect of Configuration Improvement on Option Three Performance.

The performance results of these possible modification are summarized in Figures 13, 14, and 15 for option three. Figure 13 shows that the amount of computational power increases at a much faster rate when improved reconfiguration is used. Figure 14 shows that the performance of each FPGA becomes significantly larger in smaller networks. Figure 15 shows that reconfiguration times are not as dominant for even small networks. Comparing Figures 13 and 14 to Figures 9 and 10 respectively, it can be seen that reconfiguration improvements cause option three performance to be superior for networks using as little as three XC3090s. Note that in Figures 13, 14, and 15 all four configuration possibilities shown asymptotically arrive at the same values in the limit (> 1000 XC3090s). It is no surprise that improved reconfiguration time would increase the feasibility of run-time

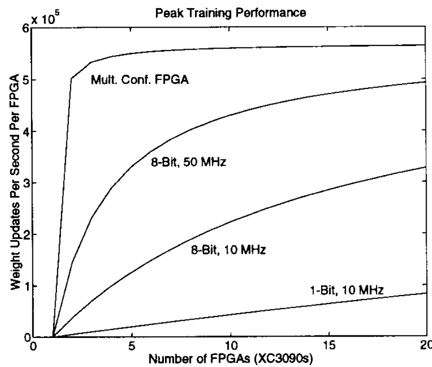

Figure 14: Effect of Configuration Improvement on Option Three Performance per XC3090.

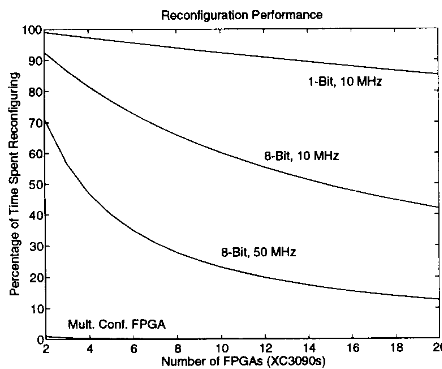

Figure 15: Effect of Configuration Improvement on Option Three's Percentage of Time Spent Reconfiguring.

reconfiguration.

5 Conclusion

The catalyst for this project was to find ways to use the FPGA's flexibility beyond prototyping. The focus was on exploiting reconfigurability–the biggest advantage a reconfigurable FPGA has over other forms of ASICs. Specifically, neural networks were used as an application, and run-time reconfiguration was used to improve neuron densities.

The Run-Time Reconfiguration Artificial Neural Network is an original neural network architecture designed and built by the authors to exploit the reconfigurability of FPGAs. RRANN implements the backpropagation algorithm by dividing it into three separate stages of execution and executing one of these stages at a time. Each stage is implemented with its own circuit module, and only one circuit module is configured onto the FPGAs at a time. When one stage completes, the FPGAs are reconfigured with the next stage's circuit module. Because circuitry from inactive stages are not using FPGA resources, the active stage's circuitry has more resources available to it. These extra resources are used to increase the number of hardware neurons on each neural processor.

This increase in available FPGA resources comes with the overhead of reconfiguration time (about 21 ms per pattern). This overhead dominates execution time when small numbers of FPGAs are being used. Two other options for implementing the backpropagation algorithm have been investigated and used for comparison. These other options require less reconfiguration overhead and achieve lower neuron densities.

Option three is able to implement 500% more hardware neurons per FPGA compared to option one. Because the computation time of the RRANN architecture increases linearly and reconfiguration time remains constant, reconfiguration overhead becomes increasingly less significant as the number of FPGAs grows. In the limit (> 1000 FPGAs), this overhead becomes insignificant, and the increased FPGA utilization becomes dominant. Moreover, in the limit, a given FPGA provides 500% better performance (WUPS) with run-time reconfiguration. If better methods for reconfiguring FPGAs were used, this better performance would be possible for even small numbers of FPGAs.

Run-time reconfiguration is another realization of the time/space trade-off common in computer engineering. This trade-off normally allows increases in performance at the cost of additional hardware or reductions in hardware at the cost of degraded performance. This trade-off applies to FPGAs and run-time reconfiguration in a slightly different sense. When more hardware is needed, the same space on an FPGA can be reused many times through reconfiguring, but doing so reduces the amount of time that the FPGA can spend executing. For example, if ten XC3090s are available, ten neurons per layer are the limit for option one. If more are needed, the same FPGAs can be reused for different stages creating more *space* for hardware neurons, but less *time* is available for computation due to reconfiguration overhead.

This new dimension to the time/space trade-off provided by the reconfigurability of FPGAs is also flexible. This trade-off does not have to be made at manufacture-time, but can be made optimally at run-time. For example, let's say a neural network with 25 neurons per layer is needed, and seven XC3090s are available. Option one is clearly insufficient providing only seven hardware neurons, and option three leaves 17 hardware neurons unused. However, option two provides 28 hardware neurons leaving only three unused, and it requires less reconfiguration overhead than option three. Since option two seems to be the optimal choice for this particular network, the decision to execute option two is made at run-time. After the completion of this 25 neuron per layer network, suppose a six neuron per layer network is needed. The run-time decision to execute option one (being clearly better for this job) can be made, and the same FPGA hardware can be reused. The flexibility of the reconfigurable FPGA can therefore be exploited on a case-by-

case basis at run-time to achieve optimal performance.

References

[1] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective*, 2nd ed., pp. 399-403, Reading, MA, Addison-Wesley Publishing Co., 1993.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel and Distributed Processing*, Vol. 1, Cambridge, MA, MIT Press, 1986.

[3] T. Watanabe, et. al., Neural Network Simulation on a Massively Parallel Cellular Array Processor: AAP-2, *IEEE International Joint Conference on Neural Networks*, 2:155-61, Washington DC, 1989.

[4] D. Hammerstrom, A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning, *IEEE International Joint Conference on Neural Networks*, 2:537-544, San Diego CA, 1990.

[5] S. S. Erdogan and T. H. Hong, Massively Parallel Computation of Back-Propagation Algorithm Using The Reconfigurable Machine, *World Congress on Neural Networks '93*, 4:861-4, Portland OR, 1993.

[6] J. Eldredge, *FPGA Density Enhancement of a Neural Network Through Run-Time Reconfiguration*, Masters Thesis, Dept. Electrical and Computer Engineering, Brigham Young University, Dec. 1993.