Predictive Control of Opto-Electronic Reconfigurable Interconnection Networks Using Neural Networks*

Majd F. Sakr[†,‡], Steven P. Levitan[‡], C. Lee Giles[†],
Bill G. Horne[†], Marco Maggini[*], Donald M. Chiarulli[⁂]

[†]NEC Research Institute
4 Independence Way
Princeton, NJ 08540
sakr | giles | horne
@research.nj.nec.com

[‡]University of Pittsburgh
Electrical Engineering
Department
348 Benedum Hall
Pittsburgh, PA 15261
steve@ee.pitt.edu

[⁂]University of Pittsburgh
Computer Science
Department
212 MIB
Pittsburgh, PA 15260
don@cs.pitt.edu

[*]Universit' di Firenze
Dipartimento di Sistemi e
Informatica
Via di Santa Marta, 3
50139 Firenze (Italy)
maggini@mcculloch.ing.unifi.it

Abstract

Opto-electronic reconfigurable interconnection networks are limited by significant control latency when used in large multiprocessor systems. This latency is the time required to analyze the current traffic and reconfigure the network to establish the required paths. The goal of *latency hiding* is to minimize the effect of this control overhead. In this paper, we introduce a technique that performs latency hiding by learning the patterns of communication traffic and using that information to anticipate the need for communication paths. Hence, the network provides the required communication paths before a request for a path is made. In this study, the communication patterns (memory accesses) of a parallel program are used as input to a time delay neural network (TDNN) to perform on-line training and prediction. These predicted communication patterns are used by the interconnection network controller that provides routes for the memory requests. Based on our experiments, the neural network was able to learn highly repetitive communication patterns, and was thus able to predict the allocation of communication paths, resulting in a reduction of communication latency.

1.0 Introduction

Communication latency is a significant issue in the design of large scale multiprocessor systems. Point-to-point interconnection networks, which directly connect all processors and/or memories, provide minimum communication latency but suffer from high cost and limited scalability. A plethora of electronic single-stage and multi-stage networks have been proposed, designed and built [Siegel90, Leighton93]. An alternative is the use of opto-electronic reconfigurable interconnection networks which offer a limited number of high bandwidth communication channels configured on demand, to satisfy the required communication traffic [CLMQ94b]. A network controller determines the network configuration based on processor requests. Once the controller provides the optical communication paths requested, the communication proceeds at high speeds. Hence, the end-to-end latency incurred by such networks can be characterized by three components: control time, which is the time needed to determine the new network configuration and to physically establish the paths; launch time, the time to transmit the data into the network; and fly time, the time needed for the message to travel through the network to its final destination. For high bandwidth short distance networks, the control time dominates the overall

*Published in the 2nd International Conference on *Massively Parallely Processing Using Optical Interconnections*, pp. 326-335, IEEE Computer Society Press, Los Alamitos, CA, October, 1995.

latency. Therefore, in order to benefit from using an optical network with high speed channels, reducing control latency is essential.

While we strive to reduce communications latency in general, and control latency in particular, it is also possible to reduce the *effects* of communication latency with techniques generally known as latency hiding. One technique is the use of locality in the communication traffic to amortize the cost of establishing a single communication path over a large number of data transfers. An implementation of that technique, called state sequence routing, is explained below. However, that technique still incurs the latency of establishing the initial communication paths for each group of messages. Our goal is to employ predictive techniques to hide the latency of establishing the initial communication paths. These techniques learn the patterns of communication and use that information to predict the need for a communication path in advance. Hence, the network provides the required communication path before a request for a path is made.

In this paper, we examine how neural networks perform at predicting the allocation of communication paths for a reconfigurable opto-electronic interconnection network in a shared memory multiprocessor environment. We chose a neural network as a prediction tool because it is well studied, well known, and easy to implement [Haykin94]. We perform on-line neural network learning and prediction for the communication patterns of three parallel applications: *temperature propagation*, *matrix multiply*, and *1-D FFT*. The next section presents the environment of our experiment, we describe our shared memory multiprocessor model and the use of neural networks as predictors. In section 3, we discuss the organization of our experiments: extraction of communication patterns, neural network learning and prediction, and evaluation. Finally, we discuss our results and make projections about future directions of research.

2.0 The Models

2.1 State Sequence Routing in a Shared Memory Multiprocessor

In our model, a shared memory multiprocessor consists of a set of **N** processing elements, **K** memory modules and a reconfigurable opto-electronic interconnection network. Such networks offer fast high bandwidth optical channels and can be configured so that any path between two components is achievable. However, only a subset of the possible paths can be implemented at any one time. Thus, a *network router controller* is required to reconfigure the network on demand through a set of configurations which provide the communication paths needed to satisfy the current traffic. When a communication path to a memory is needed by a processor and that path does not exist in the current set of configurations, the processor issues a *communication fault* and makes a request to the network controller. The network controller receives requests from all faulting processors and proceeds to reconfigure the network to service the current outstanding requests. This control system is based on the paradigm of *state sequence routing* [CLMQ94a,-CLMQ93].

Figure 1, depicts the general structure of a shared memory multiprocessor based on this paradigm. Sets of compatible (non-blocking) paths are provided by the network in a repetitive pattern, called a state sequence. The state sequence control algorithm, which runs in the state transformer block, determines the sequence. The state generator block is responsible for broadcasting the fixed length state sequence to each of the processors and memory modules. Thus, a processor waits for the network state which contains its required path to a memory. When such a

state is detected, the processor transmits its memory request. On the other hand, if an entire sequence goes by without such a path, the processor generates a fault. In response to the fault, the controller must add the required path, possibly by removing an existing path.

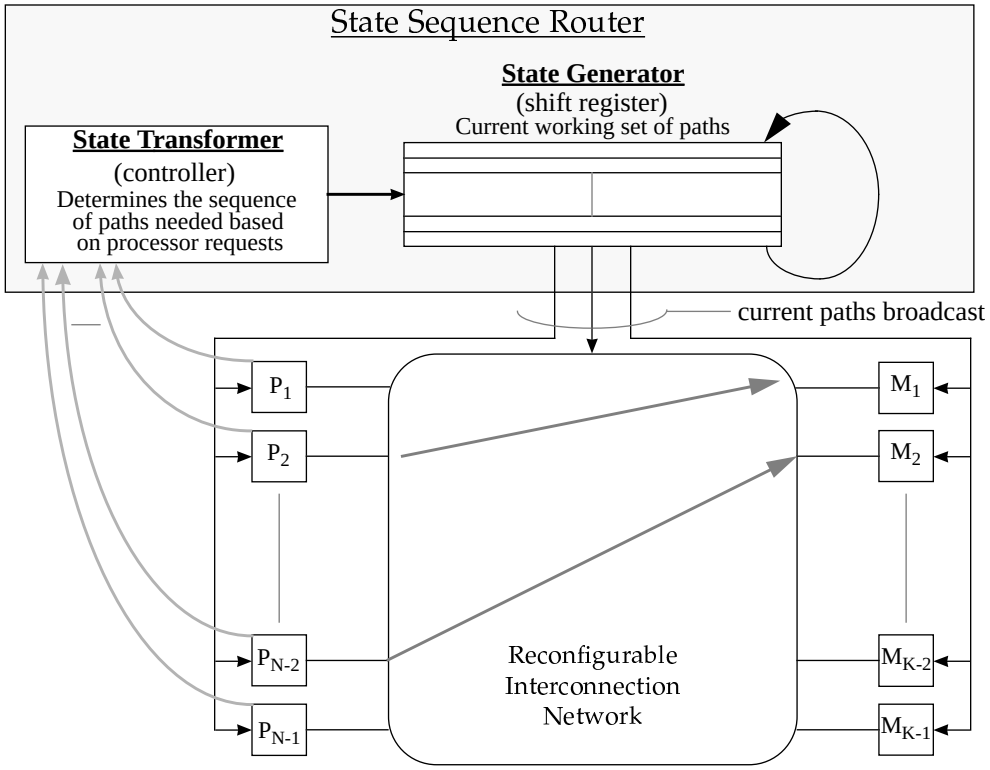

FIGURE 1. The state sequence paradigm

Since the traffic set changes dynamically as the computation progresses, the controller's task is to transform the state sequence to track these changes in the traffic. The essential point is that the control unit needs only to respond to the changes in the traffic and establish the initial paths, it is not required to respond to individual messages. Since communication patterns in a multiprocessor environment tend to exhibit locality characteristics [Johnson92], the rate at which changes occur in the traffic is lower than the message generation rate. The state sequence router exploits this locality inherent in the communication by re-using the sequence of states, or paths, repetitively.

However, the state sequence routing technique still incurs the latency of establishing the initial communication paths when responding to the changes in the traffic. This overhead can be significant when the traffic exhibits low degrees of locality such as when processes move from phase to phase in a computation. Our goal is to hide the latency of establishing these new communication paths by predicting the changes in traffic and informing the controller of a needed transformation, before the fault occurs. Consequently, the controller will transform the state sequence to include the soon-to-be-needed states thus avoiding the latency incurred by the fault.

2.2 Neural Networks as Predictors

Previously, we deduced that the traffic to be predicted changes dynamically as a function of the computation time in the multiprocessor and could be modeled as a time series. Thus, any

learning method must be able to learn a time-series and, after learning, perform as a predictor of future system behavior. Neural networks have been shown to be useful time-series predictors [Weigend93]. Many different time-varying neural networks could be used. However, for this early investigation we chose a simple input time-delay neural network (TDNN) [Lang90]. This is a feed-forward neural net with a temporal input window [Hush93].

In our multiprocessor environment, we want to train on and predict the communication patterns between the processors and the memory modules. The prediction is based on the history of behavior of the communication. The TDNN architecture with its input window possesses the memory necessary to maintain the history of the communication needed for prediction. In general these networks are trained with a supervised training procedure. A set of example input/output pairs are presented to the TDNN and a cost function of the error between the desired and actual output is minimized using, in this case, a gradient-based training algorithm [Haykin94]. Training is terminated when this error falls below some acceptable preset value. For our application, the training and prediction need to be performed on-line, meaning that the neural net must be continually re-trained as it is predicting the message traffic. This could be a problem for any neural network architecture, not just the TDNN. However, with the advent of faster and faster microprocessor chips, we expect that this will not be an issue in the near future. For the work presented in this paper, we used a simulation of the multiprocessor behavior, so that computation time for the neural network was not a factor.

3.0 Experimental Procedures

In this section, we describe our procedures for the three experiments performed. For each experiment we perform three steps. First, we use a trace driven shared memory multiprocessor simulator to generate "raw" memory traces of a parallel program. We also translate the raw traces into a set of communication patterns. Second, we use the patterns to perform on-line training of the neural network. The TDNN both trains on, and predicts, the communication patterns. Finally, we evaluate the predictions made by the neural network by using the predictions while simulating the routing of the actual messages using a simulated state sequence router and keeping a log of the number of faults incurred. The next three subsections describe in detail the steps taken in each experiment.

3.1 Extraction of Communication Pattern

A trace driven shared memory multiprocessor (SMM) simulator [Bigrigg] is used to generate the raw memory traces from the execution of a parallel program. The parallel programs use custom shared memory **load** and **store** functions. Hence, each shared memory access (load or store) is recognized by the SMM simulator which in turn writes to a file the memory accesses made by the processors. For each memory reference made, the following information is written to the trace file: the type of memory operation, the relative time between accesses for each processor, the memory address, and the value if executing a store operation. The relative time spent between accesses for each processor is measured by the SMM simulator using the UNIX "time" function. The relative time between memory accesses for a processor varies with the time that processor spends doing computation.

Because we are performing trace driven simulation, we make the assumption that the memory access time, or latency, is fixed and independent of the data. In other words, there is no pen-

alty incurred for faults, the time to access all memory is fixed. By employing trace driven simulation and making these assumptions, we lose some accuracy in the relative times of the references compared to the communication patterns which would occur in a real system. We accept these inaccuracies in this study, but note that the complexity of the task for the neural network is the same.

Each of the parallel programs used in these experiments are specifically written to run on a shared memory multiprocessor with *N=8* processors and *K=N=8* memory modules. The SMM simulator could run any *K=N* configuration, but for this preliminary investigation we maintained the same configuration for all experiments.

The raw traces generated by the simulator are stored in a non-sequential format. In order to extract the temporal communication pattern from these traces, it is necessary to serialize the traces. To perform trace serialization, we wrote a program that employs **N** memory access queues, one per processor, shown in Figure 2. Each queue contains all the memory accesses made by that processor and the relative time between memory accesses. Using the relative time between accesses for each processor, the queues are scanned to select the next memory reference. This process combines the traces to a single time line which generates sequential memory accesses. For each memory access, we record the processor number, memory access time stamp, and the memory module number being referenced.

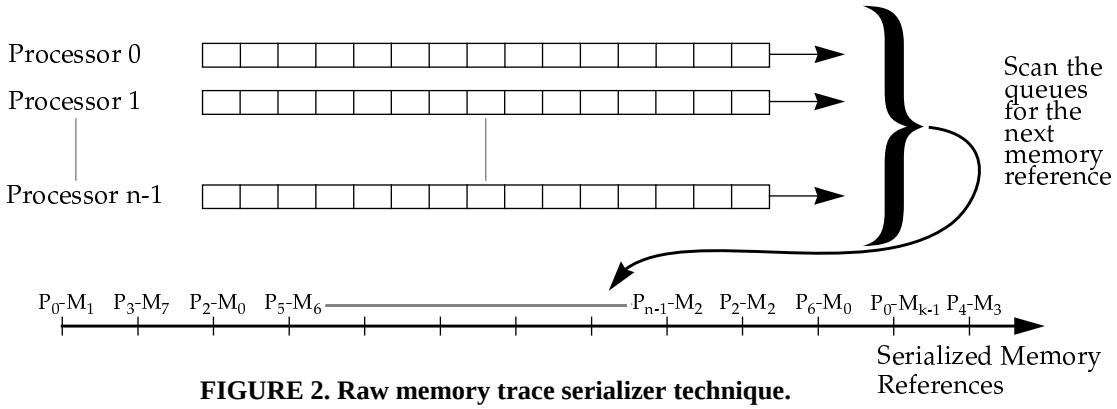

FIGURE 2. Raw memory trace serializer technique.

The serialized memory traces are then partitioned into a per processor communication pattern to simplify the observation of the communication behavior. The partitioning is performed using a technique which transforms the sequence of memory accesses to a matrix form. Using a time windowing mechanism, as shown in Figure 3, we translate the sequential memory references into a sequence of communication matrices. This technique samples the sequential memory traces and uses a fixed time window to map the memory accesses that take place in that window's time period to a communication matrix. The columns of a communication matrix correspond to the processors and the rows correspond to the memory modules. Each entry in a communication matrix describes the communication between a processor-memory pair. The construction of the matrices is binary, if any communication takes place between a processor-memory pair during a fixed time window, the appropriate entry in the communication matrix is set to one. All other matrix entries are set to zero. Because the state sequence router maintains paths in the sequence after they are used, multiple references in a single time window can be treated in the same way as a single reference.

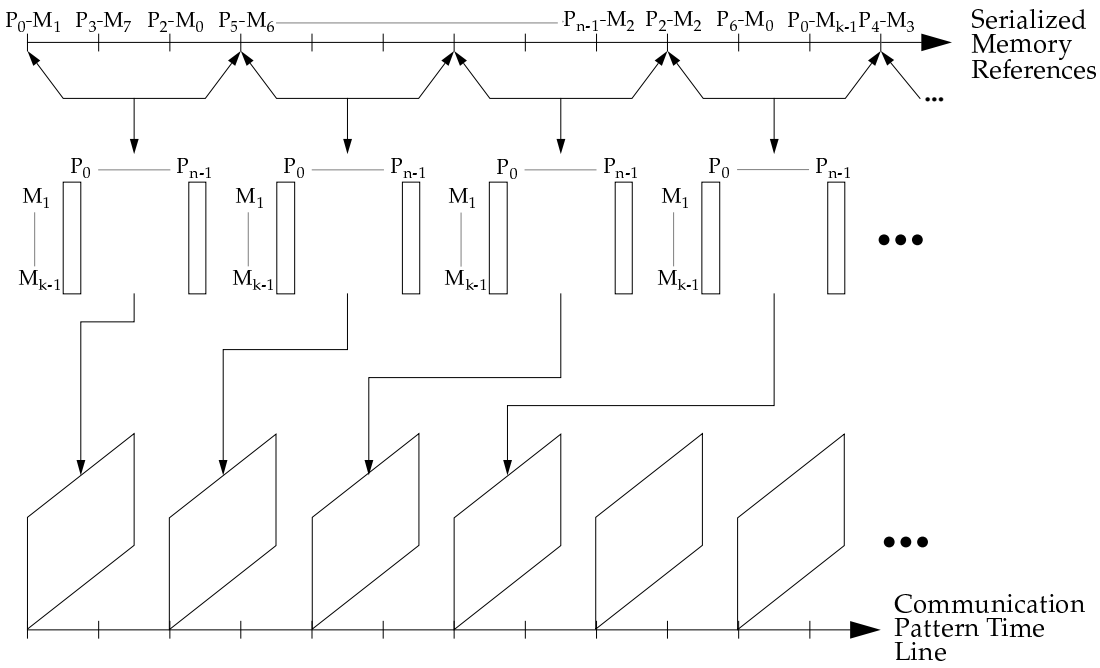

FIGURE 3. Translation of the memory references to the matrix communication pattern

Performing this time windowing, "compresses" the overall communication pattern. Therefore, the locality of references in the data is reduced, which emphasizes the changes in the communication patterns. Since our goal is to have the neural network learn and predict the *changes* in communication patterns, increasing the number of changes relative to the number of local accesses that the neural network sees, is essential. This issue is discussed in more detail below. Also, when increasing the width of the window we can have the case of a processor communicating with several memory modules in the same time window. We call these *overlapping* accesses and they must be supported, by providing multiple paths to the processor within a state sequence, during simulation.

We further simplify the learning/prediction task by using individual columns of the communication matrices as input to individual neural networks. For an NxN configuration for the multiprocessor architecture, the communication vectors are of size N. An individual neural network will train on and predict the sequence of communication vectors for each processor instead of one large neural network which uses the sequence of communication matrices. This means we will need N individual neural networks. By using the communication vectors instead of the matrices we make the assumption that the communication behavior of the processors are independent. Relaxing this assumption is the basis of future work to capture the cross processor communication.

Therefore, the sequential, compressed, one dimensional **N**x1 communication vectors of a single processor are used to perform the on-line learning/prediction of that processor's communication pattern. The next subsection describes the details of the on-line learning and prediction performed by the neural network.

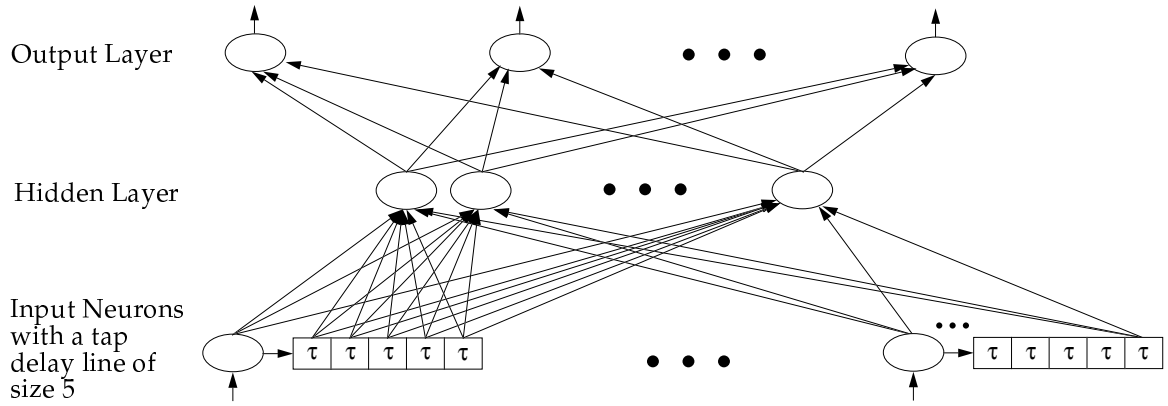

FIGURE 4. A feed forward neural network with a tapped delay line.

3.2 Neural Network Training and Prediction

In the second step of the experiments, the extracted communication patterns are used as training data for the on-line neural network prediction algorithm. Figure 4 illustrates the feed forward neural network with a tapped delay line that was used for all experiments discussed in this paper. To perform "fast" on-line training, this architecture uses conventional back propagation methods to learn the communication pattern. Furthermore, the tapped delay line maintains the history needed for learning and prediction.

The tapped delay line is a shift register of length l that acts as a set of input neurons where each entry holds a time-delayed value of the input. Hence, for m inputs the number of input neurons is $m \times (l+1)$. Therefore, as the number of inputs to the neural network increase, the size of the neural network increases vastly due to the tapped delay line. Since large neural networks train slower than smaller neural networks, we try to keep the size of the neural network as small as possible. Of course, in a real system, the neural network would be implemented as special purpose hardware which would simplify many of these issues [Sheu95].

In choosing the length of the tapped delay line we want the smallest neural network that can capture the dynamics of the system. Preliminary testing indicated that a tapped delay line of length 5 suffices to give good performance but keeps the size of the neural network small enough to perform on-line training for the applications examined in this study. This means that for our system with eight destinations, the number of input neurons is 48. The number of output neurons is also 8 since we are trying to predict the 8x1 communication vectors. We chose a hidden layer of 10 neurons which preliminary testing indicated is large enough for the problem at hand. A neural network simulator [Maggini] was used for training and prediction. The simulator uses the following parameters to control how the training/prediction is performed:

- **Prediction step**. This parameter controls the prediction frequency of the simulator. For these experiments the simulator predicts a communication vector every time step, this parameter is set to 1.
- **Training set size**. The number of input vectors that the simulator trains on every training period, 1.
- **Retrain interval**. The retrain interval parameter sets the frequency of training periods, 1.

- **Learning rate**. The learning rate controls the level of change applied to the weights after each training period. With a high learning rate the neural network will react fast to abrupt changes which is not favorable. Therefore, we set the learning rate to 0.01 so that the overall pattern of communication will be learned.
- **Maximum number of epochs**. The number of times a data set is trained on is the number of epochs of training. Training termination is determined by setting a maximum number of epochs or if the neural network's output error is less than some preset threshold. To simulate real-time training, we set the epoch number to 1.

While training, the back propagated error is calculated for an input of a communication vector at time step **t**, by comparing the neural network output (predicted) to the actual input communication vector at time **t+1**. At every time step the neural network's predicted communication vectors are written to a file. These predicted vectors are then evaluated to examine the neural network prediction performance.

3.3 Prediction Evaluation

After performing some initial training and prediction simulation experiments, we observe the following. The neural network is good at predicting the changes in the communication patterns, however, it is not very accurate about exactly when those changes will take place. We believe that this phenomena is caused by the relatively small amount of history kept in the tapped delay line, and the locality in the message traffic. However this is not a problem, since the state sequence routing mechanism can effectively use the predictions made by the neural network prior to their actual happening.

The state sequence router can take advantage of early predictions because it keeps a state in its sequence until that state in the sequence needs to be replaced by a newly needed state [CLMQ94a]. Therefore, in order to evaluate the prediction performance for our system, we use the neural network prediction to inform the router's controller to add the predicted communication paths to the state sequence. A fault occurs when a path is needed by the actual communication but not found in the router's sequence. The faults that occur while performing the routing are recorded and written to a file. Finally, the number of faults per unit time are plotted, using a time average with a large window, to show the average behavior of the system over time.

4.0 Results

In this section we discuss the results of the three experiments performed [Sakr95]. We used as input the communication patterns of following three applications. The first application is a temperature propagation program (2D relaxation algorithm); the second is a repetitive matrix multiply program; and the third is a repetitive 1D FFT program.

4.1 Temperature Propagation Application (2D relaxation algorithm)

Our first application is a simple, highly parallel program with a high degree of locality. The program is a temperature propagation/relaxation algorithm. The setup for this program is a grid of points were the temperature at each point is computed as the averaged sum of the neighboring four points. Starting with a grid of 32x32 points, initially having a temperature of zero, the temperatures of the top and right sides of the grid are set to a 100. The temperature propagates through the grid until it relaxes and the changes in temperature are below a threshold at all

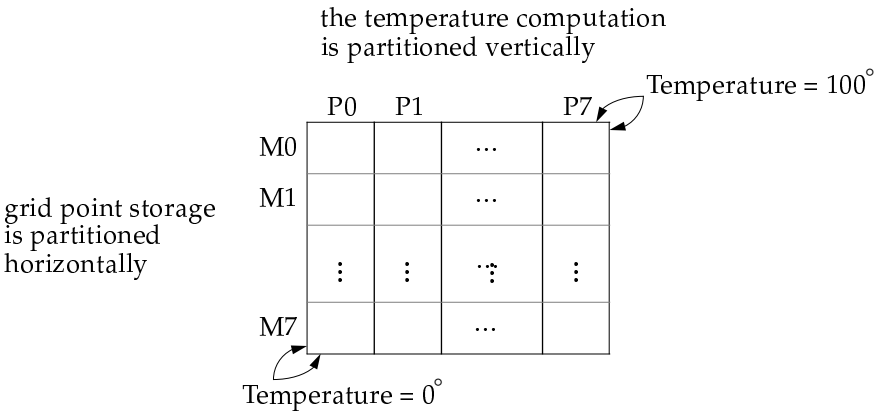

FIGURE 5. A 32x32 grid temperature propagation program, parallalized for an 8x8 multiprocessor

points. As a consequence, the program loops repetitively until the threshold is met at all points, which generates an overall repetitive communication pattern. This parallel program was written for the 8x8 SMM simulator discussed earlier. The grid is partitioned horizontally into eight sections, each stored in a separate memory module. Also, the grid is partitioned vertically into eight sections and the temperature update for the grid points in each section is computed on a separate processor. The parallelization of the program is depicted in Figure 5.

Based on the previous discussion, a time window of length 5000 is used to generate the communication matrices which emphasize the changes in the communication pattern. Using these matrices we extract the 8x1 communication vector of processor P0 and all memory modules. The overall communication pattern for this specific vector is illustrated in Figure 6b. Figure 6a shows the pattern that is repeated in this application. For this simple application, the state sequence simulator used a sequence length of 2.

The number of faults recorded by the simulation are impulses occurring over time. In order to make the data more readable, we time average the impulses and plot this average vs. time.

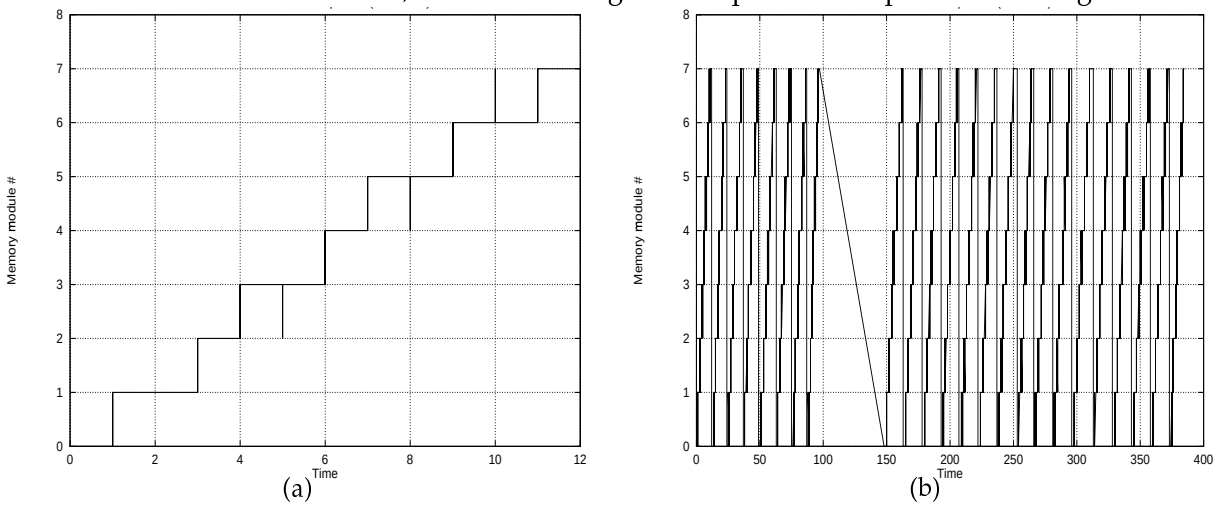

FIGURE 6. The single repetition of the communication pattern(a), and the overall communication pattern of the program(b).

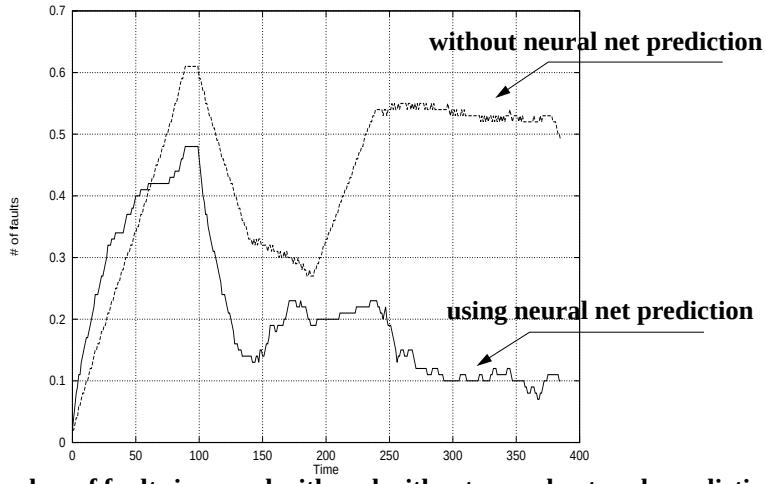

FIGURE 7. Number of faults incurred with and without neural network prediction

Figure 7 shows a plot of the number of faults vs. time occurring with and without the use of neural network prediction. Note that the initial rising slope for faults, in both plots, is an artifact of the large time window used for averaging.

In this example, the communication pattern is a stair like pattern (Figure 6), where each step depicts a new locality pattern. To be effective, the neural network has to predict these changes to new locality patterns. This is because, during each "stair step", the state sequence router satisfies all the routing needed and any prediction at that time is redundant. As Figure 7 illustrates, the neural network prediction greatly reduces the number of faults incurred during the routing of the actual communication. Hence, the neural network is predicting the changes in the communication pattern.

4.2 Matrix Multiply

As a second application we chose a parallel matrix multiply program. This program also exhibits high locality, but the communication patterns extracted are more complex than those shown for the temperature propagation program. By complex we mean that the processor alternates its accesses to the memory modules in a less uniform fashion compared to the temperature propagation program. The purpose of using this program was to investigate how well the neural network learns and predicts complex repetitive communication patterns.

The multiplication is performed on two matrices, the first is of size 16x8 and the second is 8x16 to result in a 16x16 matrix. Initially, the matrices are vectorized and concatenated into a single vector. This vector is partitioned and stored into 8 memory modules. Similarly, the computation is partitioned to the 8 processors. This procedure is depicted in Figure 8.

Using a window size of 5000, the communication pattern of a single processor (P0) and all memory modules (M0-M7) is plotted vs. time for a single matrix multiply in Figure 9a, and the overall pattern in Figure 9b. We can see that Processor P0 communicates for long periods of time with each of the memory modules. This demonstrates the high spatial and temporal locality inherent in the program. However, there is not much repetitiveness. Therefore, we repeated the matrix multiply, assuming an outer loop, to generate a repetitive pattern of memory accesses. The communication pattern also exhibits a lot of overlapping, since we used a large window size to compress the pattern. The overlapping led us to use a state sequence length of 7 to support the

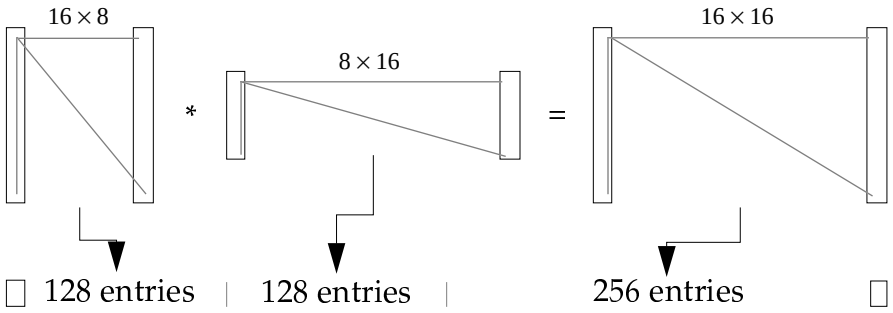

FIGURE 8. Matrix multiply program.

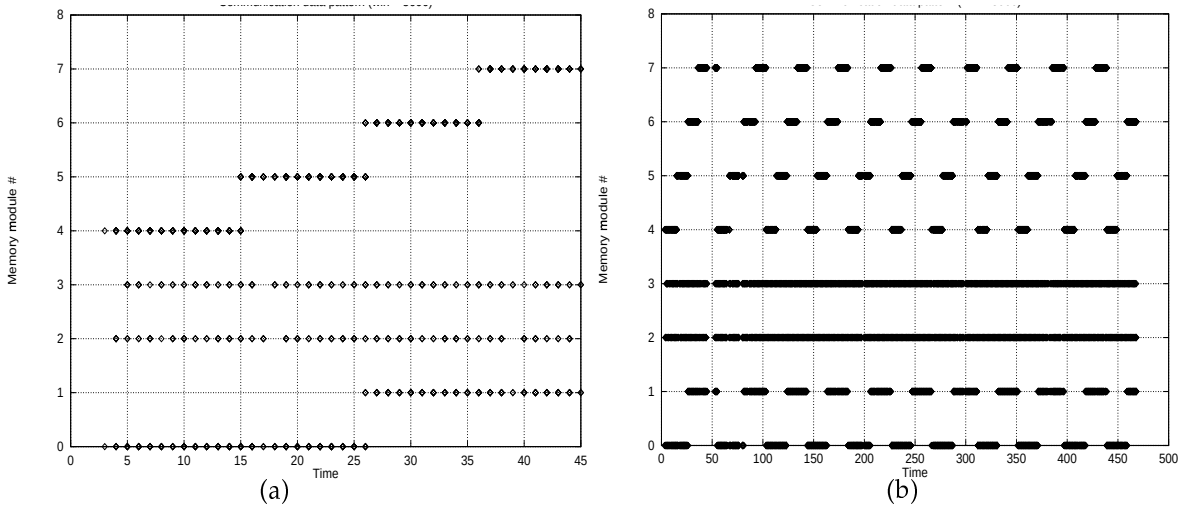

FIGURE 9. Communication pattern of a single matrix multiply (a), and the overall communication pattern of the program (b).

required memory traffic. The plot of the time averaged number of faults vs. time, Figure 10, shows that the neural network is learning and predicting these more complex patterns.

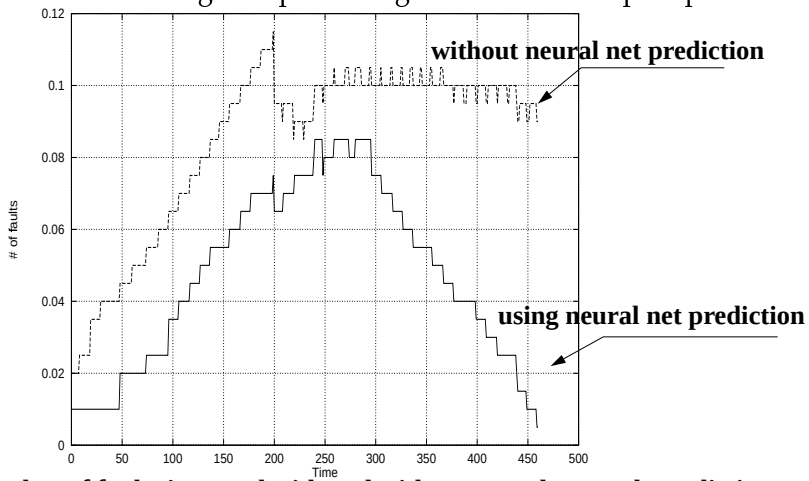

FIGURE 10. Number of faults incurred with and without neural network prediction

4.3 Fast Fourier Transform

In our third experiment we used the communication pattern generated from a parallel fast fourier transform (FFT) program. The FFT is a widely used program that is easily parallelizable [Cormen91,Hwang84]. To perform good learning and prediction, the neural network needs a repetitive pattern and a single 1D FFT does not show much repetitiveness, as shown in Figure 12a. Therefore, we decided to emulate the first half of a 2D FFT by repetitively performing the 1D FFT on an input signal with 16 sample elements. If we can show that the NN can learn and predict this pattern, then it will learn and predict the pattern of the second half of the 2D FFT.

The parallel FFT algorithm we implemented computes the FFT of a signal A with $L = 16$ samples. Therefore, $log_2\ L = 4$ steps of $L/2 = 8$ butterfly operations are needed, at each step we perform each butterfly operation on a separate processor using $L/2$ processors, as shown in Figure 11. Hence, each processor will perform the butterfly operation on the elements whose indices dif-

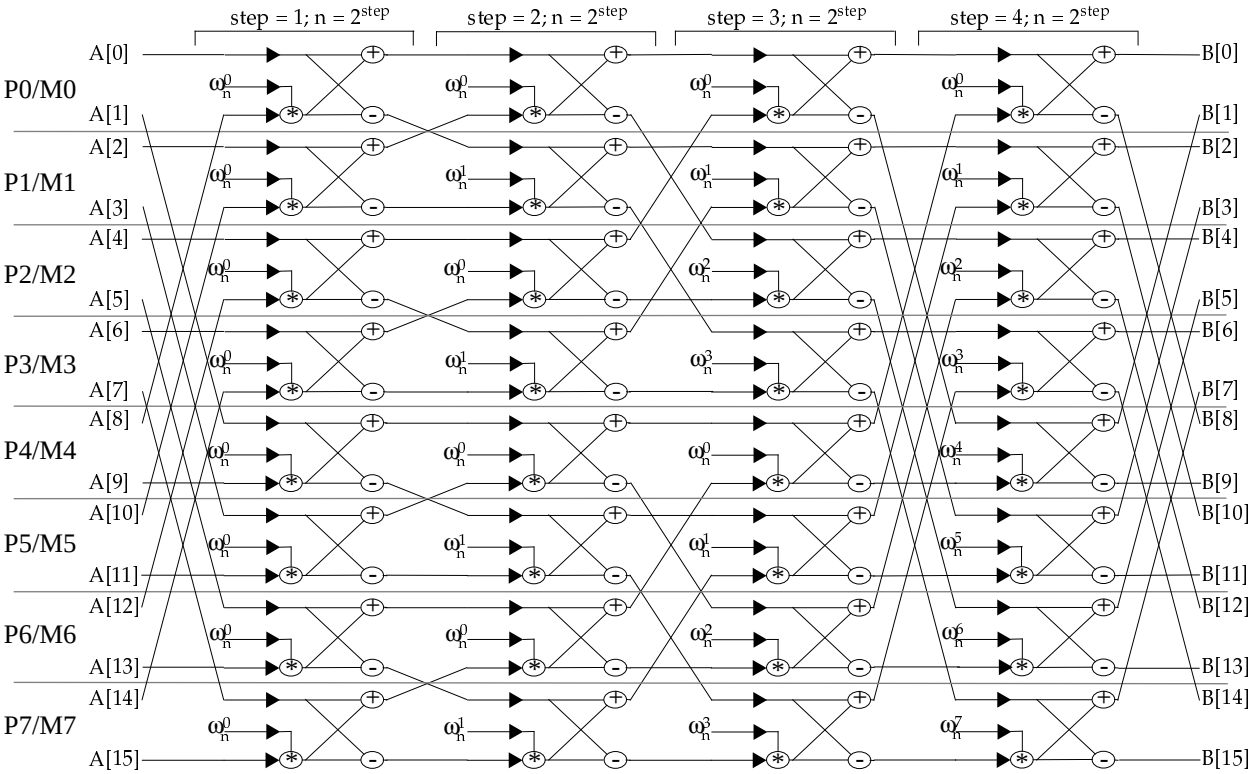

FIGURE 11. The fast fourier transform (FFT) Parallalized to 8 processors and 8 memory modules

fer in the $(4 - s)$ *-th* bit position, $s = [3, 2, 1, 0]$. The two resulting values of the butterfly, performed by processor Px, are stored at indices $2x$ and $2x + 1$respectively. The data is partitioned and mapped to the memory modules such that each memory module will hold two consecutive values of all the input, temporary, and output vectors needed. For example, in step one, processor P5 performs the first butterfly on A[3] and A[11] which were stored in memory module M1 and memory module M5 respectively. Both values resulting from the butterfly operation are stored in the temporary vector T1[10] and T1[11] in M5 in preparation for the next butterfly step. The memory access pattern of P5, while performing a single FFT, is shown in Figure 12. The FFT shows a high degree of locality to a single memory module, M5, and a low degree of locality in its other accesses, M1, M4, and M7. This is due to the fact that P5 writes all its butterfly results to

M5 and reads its input data from M1, M4, M5, and M7. Hence, the memory accesses are sparse and different, compared to the communication patterns of the other two applications discussed earlier. The communication patterns of the other processors are very similar to the one shown.

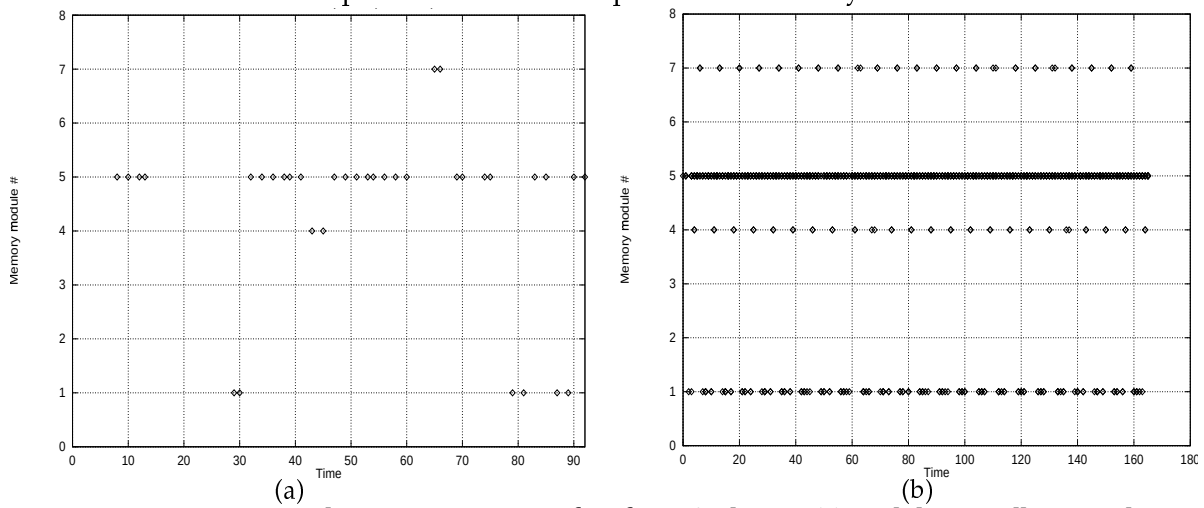

FIGURE 12. The memory accesses of P5 for a single FFT (a), and the overall repeated communication pattern (b)

Using a repetitive 1D FFT program we emulate a part of a 2D FFT program and generate the repetitive patterns shown in Figure 12b. For this example, we used a window of size 1000 time units and a sequence length of 3. The number of communication faults that occur during the routing is time averaged and plotted vs. time in Figure 13. In this example, as well, the neural network is learning and predicting the communication pattern of the FFT since the number of faults decrease with time.

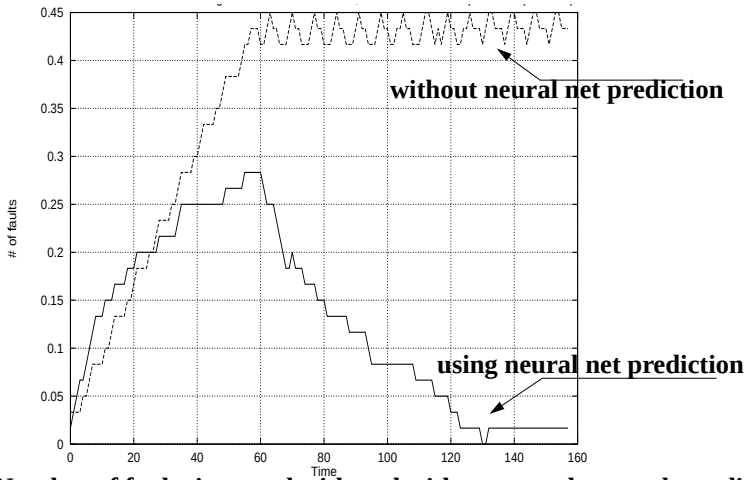

FIGURE 13. Number of faults incurred with and without neural network prediction

5.0 Conclusion

In a large scale multiprocessor environment, a completely connected interconnection network is not feasible due to cost and scalability issues. Hence, we turn to less expensive, reconfigurable, solutions that scale well but require more complex control. Using an optical reconfigurable interconnection network along with the state sequence paradigm, we can increase control efficiency and reduce latency based on the degree of locality in memory traffic. Nevertheless, this solution incurs penalties since the network controller still has to perform reconfiguration of the network for changes in the traffic. In this paper we have shown that by adding

predictive techniques along with the state sequence paradigm, this overhead is reduced due to the quality and correctness of the predictions.

For predictive techniques, we trained a time-delay neural network to learn and predict repetitive communication patterns for three applications, *temperature propagation*, *matrix multiply* and *fast fourier transform*. We make the assumption that other parallel applications have similar repetitive behaviors. The raw memory traces were translated to communication matrices using a windowing system which hides some of the locality and emphasizes the changes to a locality pattern. For the neural network to be successful in learning the changes in the traffic, we have found that we must emphasize the changes in the traffic that the neural network sees. For these three applications the neural network was able to learn and predict the change to a new locality pattern. The windowing process facilitates the learning of the communication pattern, but the predictions are made at a granularity of the window size. The state sequence router takes advantage of and uses the neural network predictions to do anticipatory reconfiguration of the optical network and thereby satisfy forthcoming communication requirements. Even if the predictions are made many time steps in the future, the state sequence router has the ability to store early predictions until their actual use. Thus, the prediction performed by the neural network provided a reduction in communication latency for applications tested.

Finally, we conclude based on this preliminary study that learning and predictive techniques which use neural networks can reduce the expensive overhead paid in performing the control of a fast optical reconfigurable interconnection network which is currently performed in a demand driven environment.

6.0 Future Work

Our short term plan is to integrate the predictions of eight neural networks to route the communication needed by all eight processors at once. Another is to investigate the use of a single neural network and attempt to learn the communication pattern using the full communication matrices instead of the vectors. Thus, we do not want to ignore the inter-processor dependencies of the communication. We would like to have a continuous simulation environment, i.e. program driven simulation instead of trace driven simulation. In a program driven simulation environment, the simulator will not ignore the fact that faults will incur more time to access memory as we do here. This will produce communication patterns which represent the real communication in a multiprocessor accurately. Also, we would like to test the performance of other neural network and machine learning architectures for this type of time series prediction. Recurrent networks are particularly attractive given the powerful real-time training algorithms, such as the extended Kalman estimators, available for training. Another interesting and open question is whether unsupervised training methods can be effectively used to aid in the communication pattern preprocessing for a supervised training method or for predicting communication latency [Goudreau94]. Finally, we would like to investigate the applicability of time series prediction techniques to the general problem of latency hiding at all levels of the memory hierarchy.

Acknowledgments

The authors would like to thank Rami G. Melhem for his helpful discussions during the early work for this paper. We would also like to acknowledge support from AFOSR Grant F-49620-93-1-0023 and NEC Research Institute.

References

[Bigrigg] M. Bigrigg, "Personal Communication".

[CLMQ94a] D. M. Chiarulli, S. P. Levitan, R. G. Melhem, C. Qiao, "Locality based control algorithms for reconfigurable interconnection networks," *Applied Optics*, vol. 33, pp. 1528-1537, March 1994.

[CLMQ93] D. M. Chiarulli, S. P. Levitan, R. G. Melhem, C. Qiao, "Bandwidth as a virtual resource in reconfigurable optical interconnections," *Optical Computing Digest*, vol. 7, pp. 229-302, 1993.

[CLMQ94b] D. M. Chiarulli, S. P. Levitan, R. G. Melhem, J. P. Teza, G. Gravenstreter, "Multiprocessor interconnection networks using partitioned optical passive stars (POPS) topologies and distributed control," *Proceedings of the First International Workshop on Massively Parallel Processing Using Optical Interconnections*, pp 70-80, April 1994.

[Cormen91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction To Algorithms," The MIT Press and McGraw-Hill, 1991.

[Goudreau94]M. W. Goudreau, C. L. Giles, "Routing in Random Multistage Interconnection Networks," *Neural Networks in Telecommunications*, Edited by Ben Yuhas and Nirwan Ansari, Kluwer Academic Publishers, 1994.

[Haykin94] S. Haykin, "Neural Networks: A Comprehensive Foundation," MacMillian, 1994.

[Hush93] D. R. Hush, B. G. Horne, "Progress in supervised neural networks," *IEEE Signal Processing Magazine*, vol. 10, pp. 8-39, 1993.

[Hwang84] K. Hwang, F. A. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, 1984.

[Johnson92] K. L. Johnson, "The impact of communication locality on large-scale multiprocessor performance", *Computer Architecture News*, vol. 20, pp 392-402, 1992.

[Lang90] K. J. Lang, A. H. Waibel, G. E. Hinton, "A time-delay neural network architecture for isolated word recognition," *Neural Networks*, vol. 3, pp. 23-44, 1990.

[Leighton93] F. T. Leighton, "Introduction to parallel Algorithms and Architectures," Morgan Kaufmann, San Mateo, CA, 1993.

[Maggini] Marco Maggini, "Personal Communication".

[Siegel90] H. J. Siegel, "Interconnection Networks for Large-Scale Parallel Processing Theory and Case Studies, Second Edition," McGraw-Hill, 1990.

[Sheu95] Bing J. Sheu, "Neural Information Processing and VLSI", Kluwer, 1995.

[Sakr95] M. F. Sakr, "Predicting Multiprocessor Communication Patterns with Neural Networks," M. S. Thesis, Department of Electrical Engineering, University of Pittsburgh, PA, 15261, 1995.

[Weigend93]A. S. Weigend and N. A. Gershenfeld,"Time Series Prediction: Forecasting the Future and Understanding the Past", Addison-Wesley, 1993.