An FPGA Implementation of GENET for Solving Graph Coloring Problems

T.K. Lee, P.H.W. Leong, K.H. Lee, K.T. Chan,
S.K. Hui, H.K. Yeung, M.F. Lo, and J.H.M. Lee
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

CSPs and GENET

Constraint satisfaction problems (CSPs) can be used to model problems in a wide variety of application areas, such as time-table scheduling, bandwidth allocation, and car-sequencing [2]. To solve a CSP means finding appropriate values for its set of variables such that all of the specified constraints are satisfied.

Almost all CSPs have exponential time complexity and instances of them may require a prohibitively large amount of time to solve. Consequently, much research has been done in developing efficient methods to solve CSPs. In particular, a generic neural network (GENET) model, developed by Wang and Tsang [3], has been demonstrated to work extremely well in solving many CSPs, often finding solutions where other methods fail.

Figure 1 shows how a graph coloring problem[1] with 5 vertices (variables) and 3 colors (values) is transformed into a neural network under the GENET model. The nodes in the network are organized into *clusters*, one for each variable in the CSP. Each node in the cluster represents a possible value for the corresponding variable. Each node can be either ON or OFF. At any given time, each cluster has exactly one ON node, which represents the value currently assigned to the corresponding variable. A connection between two nodes in the neural network indicates a constraint that would be violated if both these nodes are ON. For instance, the upper left horizontal connection indicates that vertex z_0 and vertex z_1 cannot both be assigned the color 0.

Once a CSP has been transformed into a network, the steps outlined below are performed to find one of its solutions. First, each connection in the network is assigned an initial weight of 1 and exactly one random node within each cluster is turned ON. Then, each node x in the network computes its *input*, which is defined to be the sum of the weights of the connections that are incident to x and have an ON node other than x. Afterward, each cluster *updates* itself by turning ON a node with the minimum input, giving preference to the node that is currently ON.

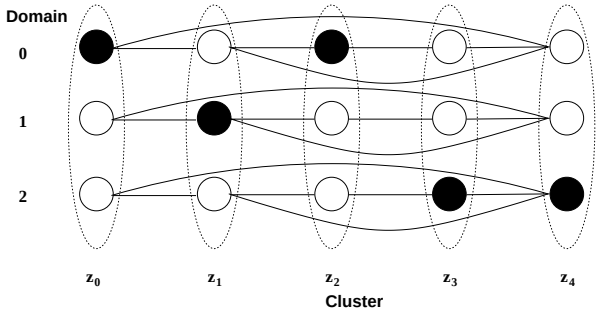

Figure 1: The GENET network (with an initial assignment) corresponding to a graph coloring problem with 5 vertices (z_0 to z_4) and 3 colors ($\{0, 1, 2\}$)

The network repeatedly performs these last two steps — compute-inputs and update-clusters — until it reaches a steady state. If no constraints are violated in that state, then a solution is reported. Otherwise, the neural net *learns* to avoid this state by incrementing the weight of any constraint that is violated there so as to decrease the likelihood of reaching that state again in the future. The network then goes back to the compute-inputs and update-clusters steps and the procedure continues until a solution is found or a preset amount of time has elapsed.

System Design

Figure 2 illustrates the overall architecture of our design for implementing GENET using FPGAs. Each GENET network is implemented as a ring of processing elements (PEs), one PE for each cluster. Each PE

[1] In the context of constraint programming, a solution for a graph-coloring problem is *any* consistent color assignment. Whether the number of colors used is minimal or not is of no importance.

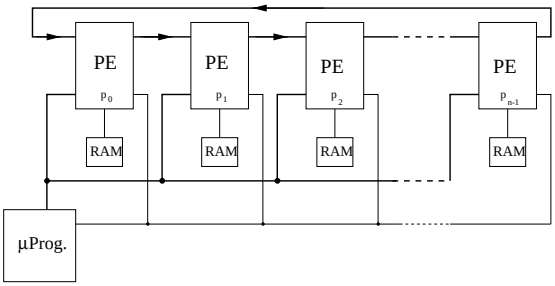

Figure 2: Overall system architecture

has access to a local memory module which contains the weights of all the connections incident to the corresponding cluster. A microprogramming unit is used to control the individual PEs.

The values of the ON nodes in all the clusters are propagated to every other cluster in the network, concurrently, in $N-1$ cycles. Consequently, each node can compute its input and each cluster can perform the appropriate update. The microprogramming unit is then informed of whether any of the PE has changed state and whether any constraint is violated. Using this information, the system then either performs the compute-inputs and update-clusters steps again or enters the learning phase, where the appropriate weights in the memory modules will be incremented.

When compared to [4], our design is more scalable as only a fixed fanin/fanout per PE is required. In addition, it offers two novel features in the field of hardware implementation of GENET. First, according to the GENET algorithm, ties for nodes with the same minimum input are to be broken randomly but fairly. In our design, this is done efficiently by executing a one-pass algorithm where the k-th candidate encountered with the current minimum input is chosen with probability $\frac{1}{k}$.

Second, it turns out that when all the clusters are updated simultaneously, the network may oscillate perpetually between two (or more) states. To prevent this occurrence, each PE, independently, has a $\frac{1}{\alpha}$ probability of cancelling an update. Consequently, we believe that we have eliminated the possibility of network-wide oscillations.

Implementation and Results

The functional specification of our design has been written in VHDL and compiled using Synopsys' *FPGA compiler* with Xilinx 4000 series FPGAs as the target platform. A PE for 25 nodes can be realized using 289 configurable logic blocks (CLBs). Using a Gigaops G900 PCB motherboard with two Xilinx XC4013E(-3) FPGAs, we have constructed a prototype system consisting of two PEs — each PE containing only four nodes due to the need to accommodate the interface circuitry — which executes successfully at 8.3 MHz. Work has begun on building a larger system.

α	No. of Updates	No. of Times Learning is Done	Total No. of Clock Cycles
2	12483	2095	2308873
4	8374	1798	1647073
8	4484	1031	898882
16	6236	1417	1246093

Table 1: Simulation results from solving g125.18

Table 1 shows the simulation results when our design is used to solve g125.18, a graph coloring problem with 125 nodes and 18 colors that is one of the standard benchmarks from the archive of the Center for Discrete Mathematics & Theoretical Computer Science (DIMACS). For a given value of α ($\frac{1}{\alpha}$ is the probability of cancelling the update of a cluster), the average statistics gathered from solving 20 instances of the problem is listed in each row.

From these empirical results, it appears that $\alpha = 8$ gives the best performance. Assuming a clock period of 120ns, our system, on average, should be able to solve an instance of g125.18 in 0.11s when $\alpha = 8$. In comparison, the average performance of published software implementations[2] of GENET ranges from 150s to 23s [1]. Therefore, a full implementation of our system is expected to produce a performance speedup of around two orders of magnitude and may provide a means to solve many CSPs that are currently intractable.

References

[1] A. Davenport, E. Tsang, C. Wang, and K. Zhu, "GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement", *Proceedings of the 12th National Conference on AI*, pp. 325-330, 1994.

[2] M. Dincbas, H. Simonis, and P. Van Hentenryck, "Solving car sequencing problem in constraint logic programming", *Proceedings, European Conference on AI*, pp. 290–295, 1988.

[3] C.J. Wang and E.P.K. Tsang, "Solving constraint satisfaction problems using neural networks", *Proceedings, IEE Second International Conference on Artificial Neural Networks*, pp. 295–299, 1991.

[4] C.J. Wang and E.P.K. Tsang, "A cascadable VLSI design for GENET", *Proceedings, Oxford VLSI Workshop*, 1992.

[2]The fastest implementation [1] uses a novel type of constraint that results in much fewer connections than our system.