A neural networks algorithm for data path synthesis

Article *in* Computers & Electrical Engineering · June 2003

DOI: 10.1016/S0045-7906(01)00047-7

1 author:

Haidar M. Harmanani
Lebanese American University
76 PUBLICATIONS **506** CITATIONS

PERGAMON

Computers and Electrical Engineering 29 (2003) 535–551

Computers and Electrical Engineering

www.elsevier.com/locate/compeleceng

A neural networks algorithm for data path synthesis

Haidar M. Harmanani *

Department of Computer Engineering and Science, Lebanese American University, P.O. Box 36, Byblos, Lebanon

Received 12 September 2000; received in revised form 1 April 2001; accepted 22 May 2001

Abstract

This paper presents a deterministic parallel algorithm to solve the data path allocation problem in high-level synthesis. The algorithm is driven by a *motion equation* that determines the neurons firing conditions based on the modified *Hopfield neural network* model of computation. The method formulates the allocation problem using the *clique partitioning problem*, an NP-complete problem, and handles multicycle functional units as well as structural pipelining. The algorithm has a running time complexity of $O(1)$ for a circuit with n operations and c shared resources. A sequential simulator was implemented on a Linux Pentium PC under X-Windows. Several benchmark examples have been implemented and favorable design comparisons to other synthesis systems are reported.

Keywords: High-level synthesis; Combinatorial optimization; Hopfield neural networks; Graph theory

1. Introduction

High-level synthesis is the design and implementation of a digital circuit from a behavioral description subject to a set of goals and constraints. From the input specification, the synthesis system produces a description of a *data path*, that is, a network of registers, functional units, multiplexers, and buses [2]. The synthesis system must also produce the specification of the *control path*. There are many different structures that can be used to realize a given behavior. One of the main tasks of high-level synthesis is to find the structure that best meets the constraints while minimizing other costs. The system to be designed is usually represented at the algorithmic level by a hardware description language such as Verilog or VHDL. The algorithmic description is parsed and represented by a data flow graph (DFG) that preserves data flow information. The nodes in a DFG represent the behavioral operations that are performed on the data (edges)

* Fax: +961-9-547256.
E-mail address: haidar@acm.org (H.M. Harmanani).

0045-7906/02/$ - see front matter
PII: S0045-7906(01)00047-7

coming into them. Directed edges leaving DFG nodes indicate data dependencies and correspond to results produced by the operations. A data path is usually synthesized in two steps, *operations scheduling* and *resources allocation*.

Scheduling assigns the operators in the DFG to control cycles or steps. The scheduling phase affects greatly *the design performance* by fixing the overall design timing and implying a lower bound on the number of resources. The lower bound on functional units is the maximum number of a given resource scheduled concurrently at a given time step. The lower bound on the number of registers is the maximum number of data flow transfers that cross the boundary of a given time step in the schedule.

Data path allocation is concerned with assigning operations and values to hardware so as to minimize the amount of hardware needed. The allocation phase is constrained by the control step schedule it implements. Thus, all operators in the scheduled DFG must be bound to ALUs; however, operators that are simultaneously active cannot share the same hardware. Variables that are active across control steps boundaries are stored in registers.

Artificial neural networks have been studied since 1943 when the first such model was suggested by McCulloch and Pitts [13]. However, Hopfield and Tank were the first to use a neural network representation for solving optimization problems [10]. The text by Takefujji [19] presents various applications of neural networks in combinatorial optimization. Though several allocation methods were proposed in high-level synthesis [2], neural networks based models have been *mostly* introduced within the context of scheduling. Within this context, Hemani's [9] scheduling method, based on Kohonen self-organizing model, is the most notable. Nourani et al. [15] proposed a scheduling method based on the Hopfield model. The method is based on moves in the scheduling space that correspond to moves toward the equilibrium point in the dynamic system space.

This paper presents a parallel algorithm to solve the *data path allocation problem*, based on the modified Hopfield neural networks (HNNs) model. We use a large number of simple processing elements or *neurons*. We assume the McCulloch–Pitts binary neuron that performs the function of a simplified biological neuron [13]. The main contributions of our method are:

- an allocation method driven by a *motion equation* that determines the neurons firing conditions,
- a parallel algorithm based on the modified Hopfield model that can solve the *clique partitioning problem*, an NP-complete problem, in constant time.

The remaining of the paper is organized as follows: In Section 2 we give some background on the HNN while in Section 3 we formulate the data path synthesis problem and describe our approach for data path optimization. Section 5 formulates the data path allocation problem using neural networks and derives the equation of motion. The parallel algorithm is described in Section 6. Results are presented and discussed in Section 7. We conclude with remarks in Section 8.

2. Hopfield neural networks: background

In 1982 Hopfield introduced the collective computational property of an artificial neural network and later proposed in 1984 a continuous output model and showed the circuit for imple-

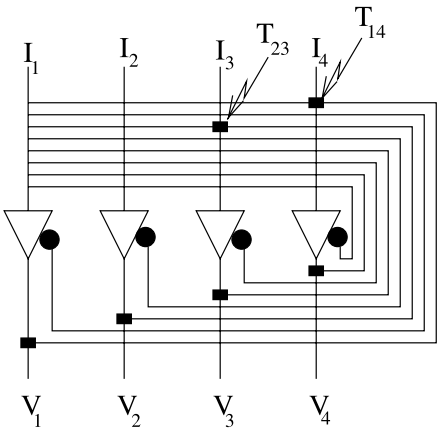

Fig. 1. Simplified Hopfield network with four neurons.

menting it. He used an NP-complete problem, the traveling salesman problem to illustrate his model.

The HNNs [10,11] are single layer networks with output feedback consisting of simple processors or neurons that can collectively provide good solutions to difficult optimization problems. A simple Hopfield network with four neurons is shown in Fig. 1. A connection between two processors is established through a conductance T_{ij} that transforms the voltage output of amplifier j to current input for amplifier i. Externally supplied bias current I_i is also present in every processor j. Each neuron i receives a weighted sum of the activation of other neurons in the network, and updates its activation according to the rule:

$$V_i = g(U_i)$$

where $g(U_i)$ can be either a binary or a threshold function for the case of the McCulloch–Pitts neurons.

Hopfield showed [10] that in the case of a symmetric connection ($T_{ij} = T_{ji}$), the motion equation for the activation of the neurons of a HNN always leads to a convergence to a stable state, in which the output voltages of all the amplifiers remain constant. The stable states of a network of N neuron units are the local minima of the energy function:

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} T_{ij}V_iV_j - \sum_{i=1}^{N} I_iV_i \tag{1}$$

where V_i is the output of the ith neuron and I_i is the externally supplied input or bias to the ith neuron. E is referred to as the computational energy of the system.

The equation of motion for the ith neuron maybe described in terms of the energy function E as follows:

$$\frac{\mathrm{d}U_i}{\mathrm{d}t} = -\frac{U_i}{\tau} + \sum_{i\neq j} T_{ij}V_j + I_i \tag{2}$$

where $\tau = RC$ is the time constant of the RC circuit connected to neuron i.

Takefujji and Lee [18] showed that the above function performs the parallel gradient descent method. In fact, as long as the motion equation of the binary neurons is given by Eq. (2), the energy function E monotonically decreases. The state of the neural network is *guaranteed* to converge to the local minimum under the discrete numerical simulation.

3. Data path synthesis

Our allocation method uses a parallel algorithm and starts with a scheduled data flow graph (SDFG) description of a circuit that has been constructed from a behavioral VHDL. The scheduling can be accomplished using any method reported in the literature [17]. In what follows, we describe our method in reference to the simple SDFG shown in Fig. 2.

Given a SDFG, a node corresponds to (1) an operator that must be assigned to a functional unit during the control step in which it is scheduled; (2) a variable that must be assigned to a register for the duration of its life time. [1] Thus, overlapping life times cannot be assigned to the same register. Finally, data transfers are assigned to some path of connections, buses and multiplexers.

Consider a SDFG node associated with a variable instance V, its corresponding operation $O(V)$, and life span $L(V)$. We define the *unit*, $U(V)$, of V as the 3-tuple:

$$U(V) = [V; O(V); L(V)]$$

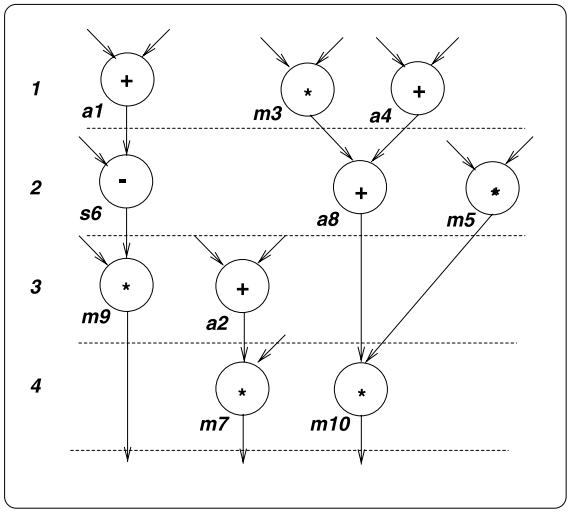

Fig. 2. Example scheduled DFG.

[1] The life span of a variable is the interval between the first time the variable was introduced and the last time it was used.

For the example in Fig. 2, the unit of $a8$, is $U(a8) = [a8; \{+\}; \{2\ 4\}]$. It follows from this definition that each SDFG node represents a unit whose output is possibly connected to the inputs of other units.

The execution of a unit $U(V)$ results in the allocation of hardware resource instances, a *functional block*. In its simplest form, a functional block includes an ALU that corresponds to the operation $O(V)$ in addition to the register allocated for the duration of $L(V)$.

The reason for introducing functional blocks is because of their direct relationship to behavioral synthesis flow graphs. In fact, the objective of our scheme is to allow the mapping of scheduled DFG nodes onto functional blocks that are the building blocks of the data path generated by the proposed allocation. The intuitive rationale is to avoid the dislocation of variable instances from their corresponding operations, keeping them close together during the entire allocation process.

Merging two or more functional blocks results in a more complex functional block characterized by an operation set that is equal to the *union* of the merged functional blocks operations. It is possible that multiplexers are introduced at the input ports of ALUs in a functional block due to a merger. The multiplexers select the appropriate operation inputs based on the clock cycle.

There are three conditions that should be satisfied for the successful merger of two functional blocks:

- There is no conflict in the use of the functional blocks operators. That is, they are assigned to different time steps in the schedule.
- There is no overlap in the registers life spans.
- The merger results in a feasible module. For example, if the resulting ALU has an operation set such as $\{+, *, /\}$ that does not exist in the technology library then this merger cannot take place from the technology library standpoint.

The reason for the second condition is to avoid resource conflicts that occur from merging two conflicting variables into the same register. The effect of this constraint could be alleviated through the insertion of *slack* nodes into the scheduled DFG. The slack nodes will be inserted so that to split the long life spans that may prevent further merging. Note that the slack nodes will eventually correspond to NO-OPs that do not add any additional operation merging cost. This technique has been used and proved to be very effective in the design examples (Section 7).

Two functional blocks that can be merged under the above conditions are called *compatible*. Compatibility relations among DFG nodes are illustrated using a *compatibility graph*, $G_{\text{comp}}(V, E)$, that consists of vertices V denoting operations and edges E denoting the compatibility relation between DFG nodes. The compatibility graph of Fig. 2 is shown in Fig. 3.

4. Optimization

The allocation method maps DFG nodes onto functional blocks. Thus, an initial data path structure (IDP) that corresponds to an initial design point is constructed. The IDP is easily generated by direct mapping of the SDFG units into functional blocks. The initial data path for the example in Fig. 2 is shown in Fig. 4. To reduce the cost of the synthesized data path it is

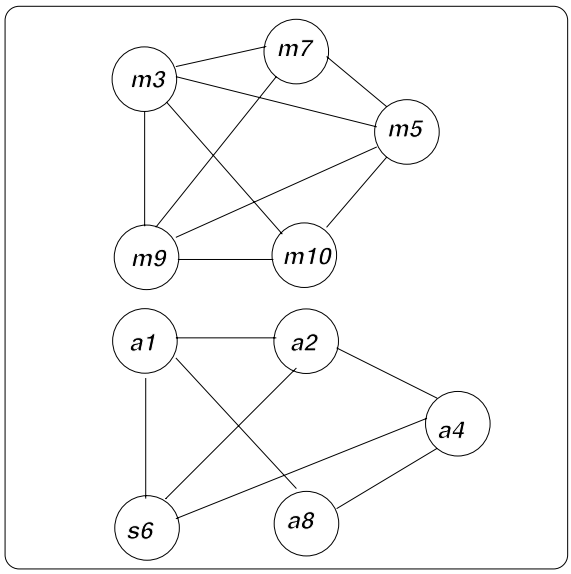

Fig. 3. Compatibility graph for the example SDFG.

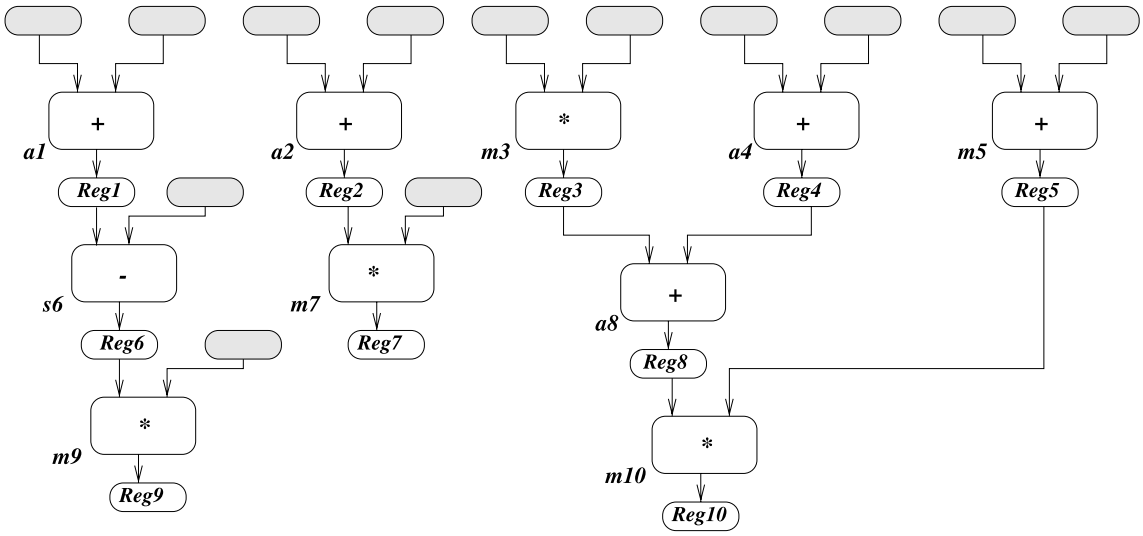

Fig. 4. Initial data path for the scheduled example DFG.

desirable to find the minimal number of functional blocks of the given SDFG. This is accomplished by merging the SDFG operations, variables and interconnects, *simultaneously*. The motivation is *twofold*. First, the method avoids the lack of merging coordination that may adversely affect the cost. For example, doing register merging first may increase the overhead of operation merging and vice versa. Second, the method leads to regularly structured data paths.

4.1. Cost considerations

Minimizing the number of components (functional units, registers,...) is not sufficient to guarantee a *good* design quality since some components may be more expensive than others under a given technology. We use a *technology library* as an input to our allocation method. This provides a good estimate of the data path cost during optimization and the computation of the cost formula. The area cost of a functional block F is approximated as follows:

$$C(F) = C(R_F) + C(O_F) + C(MUX_F)$$

where $C(F)$ is the area cost of functional block F, $C(R_F)$ is the area cost of R_F, the registers associated with functional block F, $C(O_F)$ is the area cost of O_F, the operation set of functional block F, $C(MUX_F)$ is cost of the multiplexers of functional block F.

The operation set cost $C(O_F)$ varies depending on the complexity of the functional block operation set. The register cost $C(R_F)$ is simply the cost of the registers associated with a functional block. Finally, There is an additional cost for a functional block F to account for multiplexer cost, $C(MUX_F)$, which is due to the multiplexers in the hardware implementation of F. In order to obtain a good multiplexer cost reduction, we use *register alignment*. We explain the method next.

When mapping a commutative operation to a functional block, there are two possible configurations to assign the input ports to the functional block, *left* or *right* mux. For non-commutative operations such as subtraction, the configuration is unique. When considering two functional blocks for merging, we assign the non-commutative operations to the multiplexers at the input ports of the resulting functional block first. The reason is that these signals cannot be swapped in order to explore sharing possibilities with other signals. The remaining assignment is done so as to reduce the number of multiplexer inputs, right or left, through register alignment.

For the purpose of this paper, we have derived first-cut estimates of all above cost factors based on a commercial library.

4.2. Clique partitioning problem

Let $G_{\text{comp}}(V, E)$ be a graph consisting of a finite number of compatible SDFG nodes and a set of undirected edges connecting pairs of nodes. A group of mutually compatible operations corresponds to a subset of vertices that are all mutually connected by edges. A non-empty collection C of nodes of G_{comp} forms a complete graph if each node in C is connected to every other node of C. A complete graph is said to be a clique with respect to G_{comp} if C is not contained in any other complete graph contained in G_{comp}. Therefore, a maximal set of mutually compatible operations is represented by a maximal clique in the compatibility graph. Since we can associate a functional block with each clique, the problem reduces to partitioning the graph into maximal set of compatible functional blocks. For example, the graph in Fig. 2 can be partitioned into the four cliques shown in Fig. 5(a), an optimal partitioning in this case.

The search for cliques in a graph has been proven to be NP-complete [4]. When directly solving the clique partitioning problem a solution can be determined by iteratively searching for the maximal clique of the graph and deleting it from the graph. The procedure is repeated until there are no more vertices. Thus, the problem is reduced to that of finding the maximum clique, which is

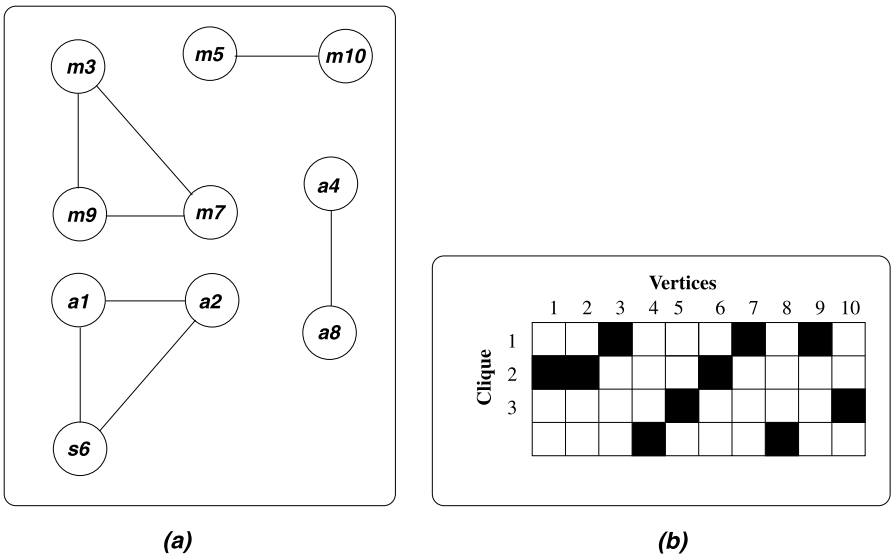

Fig. 5. Neural representation for the scheduled DFG example.

also intractable [4]. A maximum clique corresponds to a *maximum independent set* of the graph. A maximum independent set is the subset of vertices not included in a vertex cover.

The *clique partitioning problem* is solvable in polynomial time for graphs containing no complete subgraphs on three vertices using matching, and in $O(|V| + |E|)$ time for chordal and interval graphs [2]. It can also be solved in polynomial time for comparability graphs by transforming the problem into a minimum flow computation of a related network [8].

Several approximations and search algorithms were developed to solve the *clique partitioning problem*. Tseng and Siewiorek [20] presented a heuristic to solve this problem, based on the neighborhood property. The algorithm finds all cliques in a graph in a sub-optimal fashion. Bron and Kerbosch [1] find all cliques in an undirected graph using two backtracking algorithms, based on a branch and bound technique. The algorithm cuts off branches that cannot lead to a clique. This is different from the clique partitioning problem where all cliques are disjoint.

5. Allocation using neural networks

In order to solve the data path allocation problem, we formulate it as a *maximal clique partitioning* problem using the HNN model. Clearly the solution for this problem has two requirements or constraints:

- first, a SDFG node can appear in one and only one functional block (clique),
- second, all $(operation, variables)$ pairs that share the same functional block must be compatible; i.e., must constitute a clique. There is an additional requirement in order to guarantee the problem optimality and that is the number of overall shared resources must be minimal and of least cost.

We formulate the *clique partitioning problem* using a neural representation that uses a two dimensional array composed of $n \times c$ neurons where n is the total number of units and c is the number of the shared functional blocks, which are distinct cliques in the graph. A row in the network corresponds to a functional block while a column corresponds to a unit. Every array cell corresponds to a neuron connected to every other neuron using a conductance. A neuron in the array fires if it is to be a part of a specific shared resource. The firing rules in the network are determined using an energy function that will be described next. For example, our neural representation for the solution of the graph in Fig. 2 is shown in Fig. 5(b). Neurons (1,3), (1,7), and (1,9) have fired in order to share the same resource, that is *clique* 1. Other cliques in the solution are {(2,1), (2,2), (2,6)}, {(3,5), (3,10)}, and {(4,4), (4,8)}. The original graph, shown in Fig. 5(a), has 10 vertices and can be partitioned into four maximal cliques, which is the optimal solution for this example.

5.1. Motion equation

In order to enable the neurons to compute a solution to our problem, we must describe the network using an *energy function* that describes the motion of the ith neuron in the network. This will be done using the two kinds of forces that exist in neural networks, *excitatory* and *inhibitory*. For example, if a unit i is to be a part of a functional block j, then neuron (j, i) is encouraged to fire as the excitatory force. However, if the same unit is already a part of another functional block, then it is discouraged to fire as the inhibitory force.

In order to fulfill the first requirement in Section 5, we encourage one and only one neuron to fire in each column of our neural network representation, shown in Fig. 5(b). This can be fulfilled using the following term in the motion equation:

$$\sum_{j=1}^{\text{cliques}} V_{xj} - 1 \tag{3}$$

The second requirement is fulfilled by ensuring that units or neurons that fire in the same row are all compatible (fully connected). This is enforced using the graph adjacency matrix and can be illustrated using the following term in the motion equation:

$$\sum_{y=1}^{\text{nodes}} \sum_{i=1}^{\text{cliques}} C_{yx} * V_{yi} * (1 - \text{Adj}_{yx}) \tag{4}$$

where Adj is the adjacency matrix representation for the SDFG and C is the cost of the resulting functional block.

The above two terms in Eqs. (2) and (3) ensure that the network converges to a local minimum. In order to ensure an optimal solution, the system must be able to escape local minima and to converge to global minima. This is fulfilled using an additional *hill-climbing* term in the motion equation. The hill climbing term is activated only when all V_{xi} are zero. It is excitatory and will encourage the most heavily connected node in addition to edges with the maximum number of neighbors to fire first. The hill climbing term is:

$$\prod_{y=1}^{\text{nodes}}(1-V_{xy})\sum_{j=1}^{\text{nodes}}(\text{Adj}_{jx}+M) \tag{5}$$

where

$$M=\text{Max}\left(\sum_{k=1}^{\text{nodes}}\sum_{l=1}^{\text{nodes}}\sum_{m=1}^{\text{nodes}}\text{Adj}_{kl}[\text{Adj}_{km}=1][k=x \text{ or } l=x][\text{Adj}_{km}=\text{Adj}_{lm}]\right)$$

Based on Eqs. (3)–(5), the equation of motion for the ith neuron can be described by:

$$\frac{\mathrm{d}U_{xi}}{\mathrm{d}t}=-A\left(\sum_{j=1}^{\text{cliques}}V_{xj}-1\right)-B\left(\sum_{y=1}^{\text{nodes}}\sum_{i=1}^{\text{cliques}}C_{yx}*V_{yi}*(1-\text{Adj}_{yx})\right)+C\prod_{y=1}^{\text{nodes}}(1-V_{xy})$$
$$\times\left(\sum_{j=1}^{\text{nodes}}\left(\text{Adj}_{jx}+\text{Max}\left(\sum_{k=1}^{\text{nodes}}\sum_{l=1}^{\text{nodes}}\sum_{m=1}^{\text{nodes}}\text{Adj}_{kl}[\text{Adj}_{km}=1][k=x \text{ or } l=x][\text{Adj}_{km}=\text{Adj}_{lm}]\right)\right)\right) \tag{6}$$

where A, B, and C are connection weights and the bracketed notation $[S]$ is 1 if S is true and 0 otherwise.

6. The parallel algorithm

We have implemented a data path parallel algorithm using the following procedure, based on the first order Euler method. The algorithm, as will be shown later, has a running time complexity of $O(1)$.

Step 1: Initialize. Read the scheduled DFG and the technology library. Create the schedule and construct the corresponding initial data path through direct mapping of the SDFG nodes into functional blocks.
Step 2: SDFG analysis. Perform life time analysis and generate the compatibility graph. Store in G_{comp}.
Step 3: Initialize the neural network. Set $t=0$; $\Delta t=1$; $A=B=C=1$. Randomize the initial values of $U_{xi}(t)$ in the range $(-\omega,0)$ where $x=1,2,\ldots,n$ and $i=1,2,\ldots,n$ and $\omega=120$.
Step 4: Evaluate $V_{xi}(t)$ based on the binary function

$$V(X_i(t))=f(U_{xi}(t))=\begin{cases}0 & \text{if } U_{xi}\leqslant 0\\ 1 & \text{otherwise}\end{cases}$$

Step 5: Use the motion equation in Eq. (6) to compute $\mathrm{d}U_{xi}(t)$.
Step 6: Compute $U_{xi}(t+1)$ using the first order Euler method. $U_{xi}(t+\Delta t)=U_{xi}(t)+\Delta U_{xi}(t)\Delta(t)$ where $x=1,2,\ldots,n$ and $i=1,2,\ldots,n$ and $\Delta U_{xi}=\mathrm{d}U_{xi}/\mathrm{d}t$.
Step 7: Update. Increment t by 1 and update the data path cost and configuration. If the system is in equilibrium or $t=T$ then terminate. Else go to step 4.

7. Results

We have implemented the parallel algorithm based on the first order Euler method in a sequential version for verification purposes. Whereas in the parallel algorithm the output values of the neurons are simultaneously updated outside of the motion equation loop, in the sequential simulator, the output value of each neuron is individually computed as soon as the input of the neuron is evaluated inside the motion equation loop. The algorithm was implemented using the C language with a graphical interface under X-Windows on a Pentium 266 running Linux 2.2.14.

We validate our method using four high-level synthesis benchmark circuits. The benchmark examples were synthesized and compared with various synthesis systems such as the genetic algorithm approach of PSGA_Synth [3], the stepwise refinement approach of HAL [17], the simulated evolution approach of Ly and Mowchenko [12], the branch and bound connectivity based approach of Splicer [16], and the optimum linear programming based systems (LP model) in Ref. [7] and OASIC in Ref. [5]. For all attempted circuits, the system found the best results as implied by the schedule. The method is very fast and all reported results were produced in at most 1.87 CPU minutes including scheduling time.

For every example, we ran the simulator for 1000 times. At each simulation run, the network was randomly initialized, using a random seed, in the range 0 to −120. The network was able to partition all attempted graphs into the optimum number of cliques. This is consistent with the HNNs that deterministically converge to a "minimum energy" after a series of iterations.

For every benchmark, we show design details that include the type of ALUs used in the allocation process, the number of clock cycles in the scheduled DFG, the type and number of data path resources, as well as the number of edges and the number of cliques in the compatibility graph. Furthermore, we show the average number of iterations at which the algorithm converged to the optimum solution in 1000 random different runs. We notice in the results that different numbers of iterations were obtained at different simulation runs. The reason is that convergence from a given state is deterministic but starting states are different due to a different initial choice of U_{xi}. Since all graphs had many possible minima, the system converged at each simulation run to the minima that is closer to its starting state. On the average, all designs converged to an optimum solution in less than 526 iterations. Fig. 6 illustrates the number of iterations at which the simulator has converged to the optimum solution for the differential equation example out of 2500 different runs. The graph shows that the simulator converged to the optimum solution, for this example, in less than 152 iterations. Fig. 7 shows the convergence rate for the same example as the number of iterations is gradually increased for 2500 runs. Finally, Fig. 8 shows the CPU time the simulator required to converge to an optimum solution for the same 2500 different runs. For this example, the simulator was able to converge to an optimum solution for 2500 runs in less than 0.33 s in the worst case. We discuss next the benchmark examples used to illustrate our method.

7.1. The facet example

This is one of the earliest examples, and was first introduced by Ref. [20]. The initial DFG has eight nodes while the compatibility graph has 30 edges. We generated two bindings for this example, based on two different libraries. Design details are shown in Table 1. We compare our results to Facet [20], HAL [17], and Splicer [16]. Table 2 summarizes the comparison results and

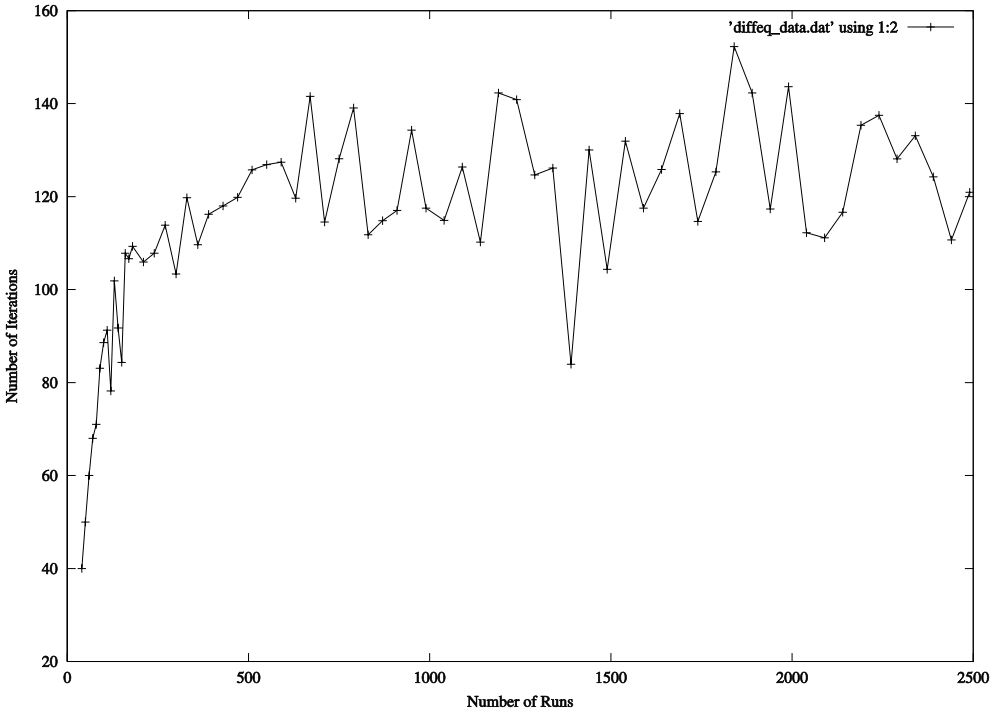

Fig. 6. Number of iterations for the differential equation example.

shows that our method, labeled neural model, achieved optimal results for this example. It should be noted that the relatively superior solution by Splicer [16] is due to the fact that Splicer is based on a branch and bound method geared toward connectivity optimization.

7.2. The differential equation example from HAL

This example solves a second order differential equation, and was first introduced by Ref. [17]. The initial DFG has 10 nodes and the compatibility graph 30 edges. Design details are shown in Table 3. Table 4 shows results comparisons to HAL [17], and Splicer [16]. Our system generated an optimum solution with respect to ALUs and registers.

7.3. Fifth-order digital elliptic wave filter

This example was popularized by Ref. [17]. The example has 34 nodes and consists of addition and multiplication operators only. We derive six designs based on four different schedules in 17, 18, and 21 time steps. Table 5 summarizes results for this example for a number of schedules with two different sets of assumptions. In the first set of results, adder delay is assumed to be one clock cycle and multiplier delay is assumed to be two clock cycles. In the second set of results, the same assumptions are used but now pipelined multipliers are used with a latency of one clock cycle. In each case, the allocation time changes however independently of the problem size. Tables 6 and 7

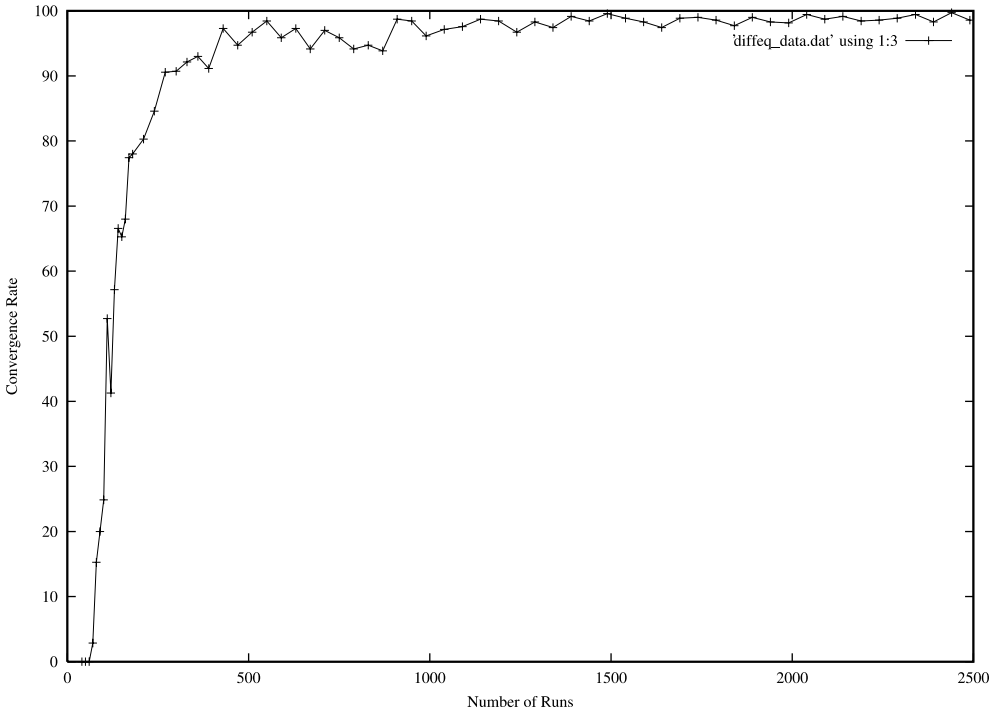

Fig. 7. Convergence rate for the differential equation example.

show results comparisons with various synthesis systems. For the elliptic wave filter example, our method achieved optimal designs in a very short time.

7.4. Discrete cosine transform

This example was reported by Ref. [14] and is used because of its large size. The discrete cosine transfer (DCT) is used extensively in image coding and compression, and has been implemented in hardware for special purpose image processors. It consists of 48 operators: 16 multiply by constant, 25 add, and 7 subtract. We generated four schedules for this example. Table 8 summarizes the allocation results for this example under the same set of assumptions used in the wave filter example. We notice that the allocation time for this example changes independently of the problem size. Table 9 shows results comparisons with various synthesis systems. In this case as well, our system found the optimal allocation for all attempted cases in a constant time.

8. Conclusion

A data path allocation method was presented based on a neural networks algorithm. The proposed system can handle pipelined and multicycled operations. The system provides a sub-optimum solution in terms of the number and types of functional units, the number of

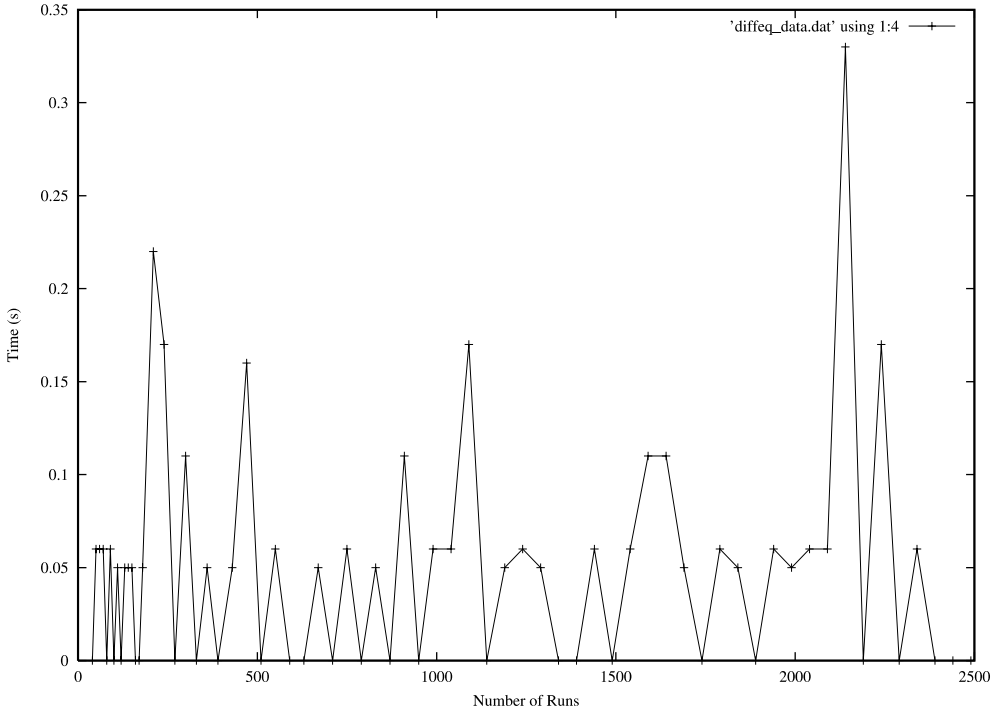

Fig. 8. CPU time per iteration for the differential equation example.

Table 1
Results details for the facet example

Design characteristics	Clock cycles	Data path resources				Graph characteristics		Average iterations	CPU time (s)
		# ALUs	Regs	Mux	Mux in	Edges	Cliques		
Non-pipelined multipliers	4	3	8	5	10	32	3	436	0.05

Table 2
Results comparisons for the facet example

System	ALUs	# Reg	# Mux	# Mux in
Neural model 1	(−&+)(∗+)(+/|)	8	5	10
Neural model 2	(/)(−&+)(∗|+)	8	7	17
HAL [17]	(/)(−&+)(∗|+)	7	6	13
Facet [20]	(/)(−&+)(∗|+)	8	7	15
Ly [12]	(/)(−&+)(∗|+)	11	5	10
Splicer [16]	(/)(−&+)(∗|+)	7	4	8

Table 3
Results details for the differential equation example

Design characteristics	Clock cycles	Data path resources				Graph characteristics		Average iterations	CPU time (s)
		# ALUs	Regs	Mux	Mux in	Edges	Cliques		
Non-pipelined multipliers	6	5	8	6	16	30	5	300	0.1

Table 4
Results comparisons for the HAL differential equation example

System	ALUs	# Reg	# Mux	# Mux in
Neural model	(∗)(∗)(+)(−)(>)	8	6	16
HAL	(∗)(∗)(+)(−)(>)	11	6	13
Splicer	(∗)(∗)(+)(−)(>)	10	5	11

Table 5
Results from the fifth order elliptic wave filter

Design characteristics	Clock cycles	Data path resources			Graph characteristics		Average iterations	CPU time (s)
		ALUs	Regs	Mux	Edges	Cliques		
Non-pipelined multicycled multipliers	17	3(∗), 3(+)	9	12	698	6	205	0.28
	18	2(∗), 3(+)	9	10	698	5	260	0.71
	21	1(∗), 2(+)	10	6	706	3	136	0.22
Pipelined multipliers	17	2(∗), 3(+)	9	10	698	5	526	0.93
	18	1(∗), 3(+)	9	8	698	4	163	0.27
	21	1(∗), 2(+)	11	6	706	3	136	0.17

Table 6
Result comparisons from the wave (multicycled multipliers)

Clock cycles	System	ALUs	# Reg	# Mux in
17	Neural model	3(∗), 3(+)	9	39
	LP model [7]	3(∗), 3(+)	–	–
	PSGA_synth [3]	3(∗), 3(+)	10	–
	Ly [12]	3(∗), 3(+)	11	–
	HAL [17]	3(∗), 3(+)	12	–
18	Neural model	2(∗), 3(+)	9	35
	LP model	2(∗), 2(+)	–	–
	PSGA_synth	3(∗), 2(+)	9	–
	Ly	2(∗), 2(+)	10	–
	HAL	3(∗), 2(+)	12	–
21	Neural model	1(∗), 2(+)	10	30
	LP model	1(∗), 2(+)	–	–
	PSGA_synth	1(∗), 2(+)	10	–
	Ly	1(∗), 2(+)	10	–
	HAL	1(∗), 2(+)	12	–

Table 7
Result comparisons from the wave (pipelined multipliers)

Clock cycles	System	ALUs	# Reg	# Mux in
17	Neural model	2(*), 3(+)	9	39
	PSGA_synth	2(*), 3(+)	10	–
	LP model	2(*), 3(+)	–	–
	OASIC [5]	2(*), 3(+)	10	–
	Ly	2(*), 3(+)	11	–
	HAL	2(*), 3(+)	12	–
18	Neural model	1(*), 3(+)	9	34
	LP model	1(*), 3(+)	–	–
	PSGA_synth	1(*), 3(+)	10	–
	OASIC	1(*), 3(+)	10	–
	Ly	1(*), 3(+)	11	–
	HAL	1(*), 3(+)	12	–

Table 8
Results from the DCT example

Design characteristics	Clock cycles	Data path resources			Graph characteristics		Average iterations	CPU time (s)
		ALUs	Regs	Mux	Edges	Cliques		
Non-pipelined multicy-cled multipliers	10	4(*), 4(+/−)	7	16	882	8	294	1.32
	18	2(*), 3(+/−)	11	10	1232	5	128	0.55
	25	2(*), 2(+/−)	13	8	1232	4	143	0.5
Pipelined multipliers	10	3(*), 4(+/−)	7	14	1216	7	172	0.88
	11	2(*), 4(+/−)	7	12	1216	6	152	0.77
	18	1(*), 3(+/−)	11	8	1232	4	134	0.33
	25	1(*), 2(+/−)	13	6	1232	3	371	1.04

Table 9
Result comparisons from the DCT (clock cycle 18)

System	ALUs	# Regs
Neural model	2(*), 3(+/−)	11
PSGA_synth [3]	3(*), 2(+/−)	13
Ref. [6]	3(*), 2(+/−)	14

registers, and the number of multiplexer inputs. The method was implemented on a Linux station using C and several benchmarks were attempted. The interesting part about our parallel algorithm versus "classical" algorithms for solving the same problem is that in our case, convergence time did not depend on the problem input size. Hence, the $O(1)$ execution time for our algorithm is apparent. Currently, we are implementing the algorithm on a Linux Beowulf parallel cluster.

References

[1] Bron C, Kerbosch J. Finding all cliques of an undirected graph – Algorithm 457. Communication of the ACM. 1973. p. 575–7.
[2] De Micheli G. Synthesis and optimization of digital circuits. New York: McGraw-Hill; 1993.
[3] Dhodhi M, Hielscher F, Storer R, Bhasker J. Datapath synthesis using a problem-space genetic algorithm. IEEE Trans CAD 1995;14:934–44.
[4] Garey M, Johnson D. Computers and Intractability – A guide to the theory of NP-completeness. New York: Freeman; 1979.
[5] Gebotys C, Elmasry M. Optimum synthesis of high-performance architectures. IEEE J Solid-State Circ 1992;27(3):389–97.
[6] Krishnamoorthy G, Nestor J. Data path allocation using an extended binding model. Proc. 29th DAC, 1992. p. 279–84.
[7] Gebotys C, Elmasry M. Global optimization approach for architectural synthesis. IEEE Trans CAD 1993; 12(12):1266–78.
[8] Golumbic M. Algorithmic graph theory and perfect graphs. San Diego: Academic Press; 1990.
[9] Hemani A, Postula A. NISCHE: A neural net inspired scheduling algorithm. Proc. EDAC, 1990.
[10] Hopfield J, Tank DW. Neural computation of decision in optimization problems. Biol Cybernet 1985;52:141–52.
[11] Hopfield J, Tank DW. Computing with neural circuits: A model. Science 1986;233:625–32.
[12] Ly T, Mowchenko J. Applying simulated evolution to high-level synthesis. IEEE Trans CAD 1993;12(3):389–409.
[13] McCulloch WS, Pitts WH. A logical calculus of ideas imminent in nervous activity. Bullet Math Biophys 1943;5:115–33.
[14] Nestor JA, Krishnamoorthy G. Salsa: A new approach to scheduling with timing constraints. IEEE Trans CAD 1993;12(8):1107–22.
[15] Nourani M, Papachristou C, Takefujji Y. A neural network based algorithm for the scheduling problem in high-level synthesis. Proc Euro-DAC, 1992. p. 341–46.
[16] Pangrle BM. Splicer: A heuristic approach to connectivity binding. Proc. 25th DAC, 1988. p. 536–41.
[17] Paulin PG, Knight JP. Forced-directed scheduling for the behavioral synthesis of ASIC's. IEEE Trans CAD 1989;8(6):661–78.
[18] Takefujji Y, Lee K. A near optimum parallel planarization algorithm. Science 1989;245:1221–3.
[19] Takefujji Y. Neural network parallel computing. Boston: Kluwer; 1992.
[20] Tseng C, Siewiorek D. Automated synthesis of data paths in digital systems. IEEE Trans CAD 1986;5(3):379–95.

Haidar Harmanani received the B.S., M.S. and Ph.D. degrees in Computer Engineering from Case Western Reserve University, Cleveland, Ohio, in 1989, 1991 and 1994, respectively. Since 1994, he has been an Assistant Professor of Computer Science at the Lebanese American University, Byblos, Lebanon. His research interests include high-level synthesis, design for testability, and cluster programming. He is a member of IEEE and ACM.