A Neural Network Based Algorithm for the Scheduling Problem in High-Level Synthesis

Mehrdad Nourani, Christos Papachristou and Yoshiyasu Takefuji
Department of Computer Engineering
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

This paper presents a new scheduling approach for high-level synthesis based on a deterministic modified Hopfield model. Our model uses a four dimensional neural network architecture to schedule the operations of a data flow graph (DFG) and maps them to specific functional units. Neural Network-based Scheduling (NNS) is achieved by formulating the scheduling problem in terms of an energy function and by using the motion equation corresponding to the variation of energy. The algorithm searches the scheduling space in parallel and finds the optimal schedule. The main contribution of this work is an efficient parallel scheduling algorithm under time and resource constraints appropriate for implementing on a parallel machine. The algorithm is based on moves in the scheduling space, which correspond to moves towards the equilibrium point (lowest energy state) in the dynamic system space. Neurons' motion equation is the core of this guided movement mechanism and guarantees that the state of the system always converges to the lowest energy state.

1 Introduction

High-level synthesis (HLS) of digital systems is concerned with the mapping of behavioral specifications into a register transfer level (RTL) structure. Typically, HLS is divided into scheduling and allocation. The scheduling strategy must consider timing and resource constraints as well as storage and interconnection cost. In fact, operation scheduling determines the hardware cost-speed trade-offs of a design. There are several scheduling methods in HLS. The simplest scheduling technique is As Soon As Possible (ASAP) scheduling where the operation in data flow graph (DFG) are scheduled into control steps from the first to the last. FACET [1] and MIMOLA [2] systems use this technique to find near optimal solution. One commonly used approach is list scheduling. It has been adapted in many systems such as [3] and [9]. In this scheme, ready operations are kept in an ordered list based on a local priority function and are scheduled in order into the next available control step until the number of scheduled operations exceed the number of resources. The forced-directed scheduling [8] technique uses a stepwise refinement technique based on the notion of force to determine the priority functions. Simulated annealing-based algorithm (a probabilistic hill-climbing method) and Integer Linear Programming formulation [11][10] have also been proposed to solve the problem more globally, allocation together in one process. Neural network-based models have been recently introduced. Kohonen self organizing model introduced in [4] is the most noticeable one. The main contribution of our method is in the following:

1. A new scheduling method which performs guided search in the huge state space of schedules, driven by motion equation which determines the firing condition for neurons.

2. A parallel algorithm based on the modified Hopfield model appropriate for implementation on a parallel machine.

3. Scheduling under fixed time and hardware resource constraints.

Our formulation of the problem and the four dimensional Hopfield model are quite innovative. Our method supports different practical aspects of high-level synthesis such as chaining, multicycle operations and pipelining. The movement mechanism to search the huge space of schedules is not random, like the simulated annealing methods. Our algorithm is a deterministic method and it always converges to the global optimum if there is not a time limit on the neurons relaxation process. Contrary to the simulated annealing and Kohonen's model [4] that heavily rely on the definition of the cost function and empirical constant tuning, we do not use any cost function in our model. This is done by implicitly using the energy function in the neuron's firing equation (called motion equation). Furthermore, we significantly reduce the performance sensitivity to the empirical coefficients by considering a specific term for each constraint within the motion equation.

This paper is organized as follows: Section 2 describes our neural network and neurons' motion equation. Section 3 presents NNS algorithm. Finally, the experimental results are shown in section 4.

2 Network Organization and Neurons' Motion Equation

2.1 Organization

Our method, Neural Network Scheduling (NNS), uses a four dimensional neural network. Each dimension corresponds to one of the important factors involved in the scheduling problem specifically, the time, the functional unit type, the functional unit index and the operation index in the data flow graph (DFG). Depending on the type of the optimization problem, the user may specify some constraints on time (es) and maximum number of FU's or a combination of them. To construct a suitable neural network for the scheduling problem, we consider one dimension for each of the following: 1) Time, 2) FU types, 3) Number of FU's of each type and 4) Operations. This neural network model is shown in Figure

---

This work was partially supported by the Semiconductor Research Corporation (SRC) under contract No. 91-DJ-147
1. Notice that the four dimensions are shown in two figures for convenience.

Each node in DFG corresponds to a specific four dimension hypercube. Based on the motion equation, presented in section 2.2.3, only one neuron in this hypercube is fired in each iteration and its address (four indices) provides necessary information (time, FU type and FU index) to schedule that operation. Moreover, in the Hopfield net the output of each neurons is one of the inputs to all others so, the interaction between neurons provides very global scheduling decisions. In fact, after relaxation of the neural network, we will obtain the minimum energy state (equilibrium point) corresponding to the minimum time or minimum hardware cost schedule, depending on the user’s optimization problem and constraints.

2.2 Motion Equation

2.2.1 Notation

We use the following notations in the motion equation:
- Time index expressed in control step: \( 1 \leq i \leq cs \)
- Type index: \( 1 \leq j \leq type \)
- FU index of type \( j \): \( 1 \leq k \leq \text{max}[j] \)
- Operation index: \( 1 \leq l \leq op \)
- Type of operation: \( \text{typeop}[l] \)
- Number of predecessors of operation \( l \): \( \text{numpred}[l] \)
- Number of successors of operation \( l \): \( \text{numsucc}[l] \)
- \( d \)th predecessor of operation \( l \): \( \text{pred}[d][l] \)
- \( d \)th successor of operation \( l \): \( \text{succ}[d][l] \)
- Time step of operation \( l \) in ASAP schedule: \( \text{asap}[l] \)
- Time step of operation \( l \) in ALAP schedule: \( \text{alap}[l] \)
- Input stimulus of a neuron in the network: \( U_{i,j,k,l} \)
- Output of a neuron in the network: \( V_{i,j,k,l} \)
- Difference between two inputs of a neuron in two consecutive iterations:
\[
dU_{i,j,k,l}(t+1) = U_{i,j,k,l}(t+1) - U_{i,j,k,l}(t)
\]

2.2.2 Binary Neuron and Energy Function

It is obvious that operation "i" can be performed only by a FU of its type. For example if the 8th operation is an addition and an adder is considered to be the second FU type in the design, then \( \text{typeop}[8] = 2 \) means an addition can only be performed by one of the FU's of type 2 (adder). In other words, only the neurons whose indices \( j \) and \( p \) satisfy \( \text{typeop}[l] = j \) are effective in convergence. So we can simply eliminate all other neurons because they have never been fired (i.e. their inputs and outputs are always zero). For the neurons whose indices (address) satisfy \( \text{typeop}[l] = j \), we have the following firing condition in each iteration:

\[
V_{i,j,k,l} = \begin{cases} 
1 & \text{if operation "i" is performed by} \quad \text{"k"th FU of type "j" in "i"th control step}.
0 & \text{otherwise.}
\end{cases}
\]

The above condition shows that we actually use binary neurons in our model. We will shortly change it in order to obtain better performance.

Hopfield and Tank introduced a deterministic neural network model [5] with a symmetrically interconnected network, shown in Figure 2. Each neuron is simulated by an operational amplifier that has a sigmoid input/output function defined as \( V_o = g(U_o) = \frac{1}{2}(1 + \tanh(U_o)) \), where \( U_o \) and \( V_o \) are the input and output voltage of the \( o \)th amplifier. It becomes a step function (binary neuron) if \( \lambda \gg 1 \). Conductance \( G_{o,2} \) connects the output of the \( p \)th amplifier to the input of the \( o \)th amplifier. It can be easily shown that the motion equation describing the time evolution of the analog circuit is

\[
\frac{dV_o}{dt} = -\frac{U_o}{\tau} + \sum_{l=1}^{N} G_{o,l} V_l + I_o, \quad \text{where} \quad \tau = RC
\]

is the time constant of the circuit. Hopfield has shown that the motion equation for a network with symmetric connections \( (G_{o,2} = G_{o,3}) \) always leads to a convergence of stable states, i.e. the outputs of all neurons remain constant. Furthermore, when the amplifiers are in a high gain mode, the stable state of a network comprising \( N \) neurons is the local minima of the energy:

\[
E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} G_{i,j} V_i V_j - \sum_{i=1}^{N} V_i I_i.
\]

Before discussing the details of our method, similar to what some researchers have shown [6], we will show that our neural network model can perform the parallel gradient descent method. In other words, based on the firing condition of a neuron, as long as the motion equation satisfies:

\[
\frac{dU_{i,j,k,l}}{dt} = -\frac{dE}{dU_{i,j,k,l}} + \sum_{j,k,l} \frac{dV_{i,j,k,l}}{dt}
\]

In fact, whatever the computational energy function \( E \) is, the motion equation forces it to monotonically decrease. The following proof shows that the motion equation forces the state of the system to converge to the local minimum:

\[
\frac{dE}{dt} = \sum_{i,j,k,l} \sum_{j,k,l} \frac{dV_{i,j,k,l}}{dt} \frac{dV_{i,j,k,l}}{dt} = \sum_{i,j,k,l} \frac{dV_{i,j,k,l}}{dt} \frac{dU_{i,j,k,l}}{dt}
\]

As long as the input/output function of the processing elements (firing condition of neurons) is a nondecreasing function, i.e. step function, \( \frac{dV_{i,j,k,l}}{dt} \) have to be positive or zero. Consequently, \( \frac{dE}{dt} \) is negative or zero which guarantees that the state of the system always converges to a local minimum. These facts provides two important advantages for our algorithm:

1. Although it is possible to find the exact form of energy function \( E \), we don't need it in the process because based on \( \frac{dE}{dt} = -\frac{dE}{dt} \) when we apply \( \frac{dU}{dt} \) to find a new state of the system, we implicitly use the energy function.

2. Converging to the lowest energy function is guaranteed by \( \frac{dE}{dt} \). Thus, there is no need to define and optimize any additional cost function. In fact, the optimization process is implicitly considered in the motion equation.

2.2.3 Original Motion Equation

To avoid complicating the motion equation, we first explain the basic and simple motion equation and then we describe the modified version of this equation. To introduce the motion equation, we present all of the constraints, including the user's constraints, and the data flow dependencies in a compact form:
\[
d\alpha_{i,j,k,l} = \begin{cases} 
\frac{\text{numpred}[i]}{C_i} & \text{if } x = 0 \\
-\frac{1}{C_i} & \text{otherwise}
\end{cases}
\]

where:
\[
D_i = \text{numpred}[i] ; \quad C_i = \max\{\text{typeop}\text{pred}[i][d][d]\}
\]
\[
D_i = \text{numsucc}[i] ; \quad C_i = \max\{\text{typeop}\text{succ}[i][d][d]\}
\]
and \(h(x)\) is a hill-climbing function such that:
\[
h(x) = \begin{cases} 
1 & \text{if } x = 0 \\
0 & \text{otherwise}
\end{cases}
\]

As we have already mentioned, this motion equation is valid only for the neurons whose addresses satisfy \(\text{typeop}[i] = j\). The first and second terms guarantee that the data dependency constraints (predecessors and successors relations) are satisfied. Briefly, an operation can be assigned to control step \(i\) if and only if all of its predecessors are assigned in control steps \(1\) through \(i-1\) and also all of its successors are assigned in control step \(i+1\) through \(c_s\), assuming that all operations consume only one control step. Multi-cycle operations can also be considered as several consecutive similar operations. The third term shows that only one operation should be assigned to a suitable FU in a specific control step. The fourth term guarantees that an operation is performed by the suitable FU within the possible control steps only once. The last term is a hill-climbing term to guarantee that no control step remains empty. This term may be used if a balanced distribution of operations within the \(c_s\) control steps is desired. Specifically, when we solve the optimization problem under the time constraint case, we want to use all possible control steps in order to obtain minimal hardware cost.

\(C_1, \ldots, C_s\) are experimentally pre-determined positive constants representing the relative effect (weight) of each constraint (conflict) in changing the energy of a neuron. We observed that the performance of NNS is not sensitive to the tuning of these coefficients. Our empirical sensitive analysis has shown that assigning a similar number (i.e. 1) to all of them is a good tuning. In fact, because of the nature of the constraints in scheduling, none of them should be able to dominate the effect of others.

2.3 Modifications

2.3.1 Reduction of Neurons

- The maximum number of neurons (a rough upper bound) is: \(c_s * \text{type} * \max\{\text{max}[j]\} + \text{op} = R * \text{cs} * \text{op} \in O(\text{cs} * \text{op})\). Because operation \(f\) is a specific type, we may only consider the required neurons corresponding to that type. So the number of neurons for operation \(f\) will be reduced to: \(c_s * \max[j]\) where \(\text{typeop}[i] = j\).

- We can assume that operation \(l\) can be assigned between its corresponding control steps in the ASAP and ALAP schedules. So we have: \(\text{asap}[f][l] \leq \text{time of operation } f \leq \text{alap}[f][l]\). By using this modification, the number of neurons for operation \(l\) is: \((\text{alap}[f][l] - \text{asap}[f][l]) * \max[j]\). This reduction results in significant improvement in convergence speed.

2.3.2 Impossible Conditions

- The minimum required number of control steps is determined by ASAP schedule. So if \(c_s < \max\{\text{control steps}\} \text{ in ASAP schedule}\), the algorithm should be terminated.

- If there are \(\maxop\text{type}[j]\) operations of type \(j\), then the minimum required number of FU type \(j\) is:

\[
\min[j] = \frac{\maxop\text{type}[j]}{c_s}
\]

Theoretically, the maximum number of FU type \(j\) is: \(\maxop\text{type}[j]\), so totally:

\[
\min[j] \leq \max\{\text{FU}[j]\} \leq \maxop\text{type}[j]\]

The maximum values (\(\max[j]\)) may also be specified by the user as hardware constraints and if there exist at least a type index \(j\) such that \(\max[j] < \min[j]\), the the algorithm should be terminated.

2.3.3 Increasing the Convergence Speed

- Using the maximum neuron model: In data flow graph each operation must be executed once. To guarantee this, the algorithm should fire only one neuron out of all considered \(\max[j] * (\text{alap}[f][l] - \text{asap}[f][l])\) neurons for each operation. In fact, the processing elements should behave as in the modified McCulloch-Pitts neuron model [7]. So, we will use the following maximum neuron instead of binary neuron:

\[
V_{i,j,k,l}(t) = \begin{cases} 
1 & \text{if } U_{i,j,k,l} > 0 \text{ and } U_{i,j,k,l} = \max[j](U_{a,j,k,l}) \\
0 & \text{otherwise}
\end{cases}
\]

By using this model, only one of the neurons corresponding to a specific operation will be fired so we can eliminate the fourth term in the motion equation. On the other hand, the neurons corresponding to the empty control steps have larger input values, hence they have a better chance to fire. Thus, the fifth term in the presented motion equation can also be dropped from the equation.

These modifications result in a new motion equation. This equation along with the algorithm is described in the next section.

3 Algorithm

The input of the NNS algorithm is a design behavior description, usually expressed by a data flow graph (DFG). The output of the NNS algorithm is a scheduled DFG showing the hardware modules (FU) and the control step that each operation should be performed in it. The NNS algorithm can be summarized as follows:

- **Step 0 (Initialize):**
  - Read DFG and the constraints (\(\max[j]\) and \(c_s\)).
  - Find ASAP schedule. If \(c_s < \text{total control steps in ASAP schedule}\), then terminate the algorithm with "IMPOSSIBLE" message.
  - Find ALAP schedule.
- Find min[f]. If \( \exists j \) such that \( \max[f] < \min[f] \), then terminate the algorithm with "IMPOSSIBLE" message.
- Set \( C_1 = C_2 = C_3 = 1 \) and \( t = 0 \).

**Step 1** (Randomize initial values): The initial state is ASAP schedule, but it is not unique because there are \( \max[f] \) number of FU's of type \( j \). The operations are randomly distributed among different FU's by assigning a random number to all neurons whose addresses are: \( \text{asap}_t[l] \leq i \leq \text{alap}_t[l] \), \( 1 \leq j \leq \text{type} \), \( 1 \leq k \leq \max[f] \) and \( 1 \leq l \leq \text{op} \). Briefly, \( U_{i,j,k,l}(t) = r \) where \( r = \text{random)[-20,3} \).

**Step 2** (Evaluate the outputs):

\[
V_{i,j,k,l}(t) = \begin{cases} 1 & \text{if } U_{i,j,k,l} > 0 \text{ and } U_{i,j,k,l} = \text{maz}_{a,c} \{U_{a,j,c,l}\}, \\ 0 & \text{otherwise} \end{cases}
\]

**Step 3** (Compute \( U_{i,j,k,l}(t+1) \)): Compute new inputs for neurons \( U_{i,j,k,l}(t+1) \) for all operations based on the first order Euler formula*: \( U_{i,j,k,l}(t+1) = U_{i,j,k,l}(t) + \Delta U_{i,j,k,l}(t) \) where:

\[
\Delta U_{i,j,k,l}(t+1) = -C_1 \cdot \left[ \text{numpred}_l \right] - \sum_{d=1}^5 \left[ \sum_{c=1}^C \sum_{a=1}^A \text{asap}_t[l][d][c][a] V_{a,\text{typeop}[\text{pred}[l][d]],[c][a][d]} \right] - C_2 \cdot \sum_{d=1}^5 \left[ \sum_{c=1}^C \sum_{a=1}^A \text{alap}_t[l][d][c][a] V_{a,\text{typeop}[\text{succ}[l][d]],[c][a][d]} \right] - C_3 \cdot \sum_{d=1}^5 \left[ \sum_{c=1}^C \sum_{a=1}^A \text{asap}_t[l][d][c][a] V_{a,\text{typeop}[\text{succ}[l][d]],[c][a][d]} \right] - C_4 \cdot \sum_{d=1}^5 \left[ \sum_{c=1}^C \sum_{a=1}^A \text{alap}_t[l][d][c][a] V_{a,\text{typeop}[\text{succ}[l][d]],[c][a][d]} \right]
\]

where:

- \( D_0 = \text{numpred}_l \); \( C_0 = \text{max}[\text{typeop}[\text{pred}[l][d]]] \)
- \( D_0 = \text{alap}_t[l][d][c][a] \); \( A_0 = \text{alap}_t[l][d][c][a] \)
- \( A_0 = \text{Min}[\text{alap}_t[l][d][c][a], i-1] \); \( a_0 = \text{asap}_t[l][d][c][a] \); \( i = 1 \); \( a_0 = \text{alap}_t[l][d][c][a] \);

To keep the neurons competitive with each other, we restrict their inputs as follows:

- If \( U_{i,j,k,l}(t+1) > U_{\text{max}} \) then \( U_{i,j,k,l}(t+1) = U_{\text{max}} \).
- If \( U_{i,j,k,l}(t+1) < U_{\text{min}} \) then \( U_{i,j,k,l}(t+1) = U_{\text{min}} \).

where: \( U_{\text{max}} = +20 \) and \( U_{\text{min}} = -10 \) are empirically determined constants.

**Step 4** (Termination condition): If all constraints are satisfied (i.e. all \( \Delta U_{i,j,k,l} = 0 \)) then terminate the algorithm, else increment \( t \) by one and go to step 2.

### 4 Experimental Results

The NNS algorithm described in the previous sections has been implemented in C on a SUN SPARC-SLC workstation. Extensive consideration is given to some important applications including multi-cycle operations, chained operations, structural pipelining and functional pipelining. This section presents brief results for six design examples presented in the literature.

*This is the main parallelizable formula in our algorithm. There are many parallel architectures (such as mesh, hypercube and connection machine) by which this formula can be computed in parallel [14].

The following tables give a summary of the design results produced by our method. In Table 1, we compare our method with some methods found in the literature. We only consider the results for the fifth order elliptical filter in a non-pipelined system. It contains 26 additions and 8 multiplications. As most systems do, we suppose a multiplication takes two cycles, while an addition takes only one cycle to complete. In this table, the first row represents the schedule obtained in [12] using an as soon as possible (ASAP) algorithm. The second row shows Forced-Directed scheduling result presented in [8]. The third row is obtained by the System Architect's Workbench (SAW) at CMU [13] using a list scheduling algorithm. The results of the ADPS algorithm [10] and ALPS algorithm [11] are presented in the fourth and fifth rows. Both approaches are based on linear programming formulation. Finally, the sixth row shows the result obtained by our method. Table 2 shows the result of our algorithm for the above six examples. Table 3 shows the average number of iterations required to obtain the optimal result (global minimum found by other researcher) and the frequency of convergence to this result for the above six examples in a non-pipelined system. Each example was run 100 times. If the number of iterations exceeded 1000, we would assume that the algorithm could not catch the global minimum in that run. The running time for these examples is 0.5 to 90 seconds on a SUN SPARC-SLC workstation. However, this running time does not represent full capability of our parallel algorithm, because we have done the implementation on a sequential machine. In fact, our C code on a sequential machine is just a simulation of the parallel algorithm. The running time will be reduced significantly if we use a parallel architecture.

### References


Figure 1: Four dimension neural network

Figure 2: Binary neuron (processing element) and analog computational network

Table 1: Comparison with other methods for example #6 (Elliptical Filter)
### Table 2: The results of running NNS algorithm on six examples

<table>
<thead>
<tr>
<th>Example Name</th>
<th>FU types</th>
<th>Execution time</th>
<th>Special features</th>
<th>Total number of control steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACET (l = 8)</td>
<td>*,+,-, /,&amp;,V</td>
<td>All take 1 cycle</td>
<td>T=4, T=5</td>
<td>T=4, T=5</td>
</tr>
<tr>
<td>MAHA(a) (l = 16)</td>
<td>+,-</td>
<td>Both take 1/2 cycle</td>
<td>+,- can be chained</td>
<td>T=4</td>
</tr>
<tr>
<td>HAL (l = 11)</td>
<td>*,+,-, &gt;</td>
<td>All take 1 cycle</td>
<td>T=4, T=5, T=7, T=8</td>
<td>T=4, T=5, T=7, T=8</td>
</tr>
<tr>
<td>Band Pass Filter (l = 29)</td>
<td>*,+,-</td>
<td>All take 1 cycle</td>
<td>T=8, T=9, T=11, T=13</td>
<td>T=8, T=9, T=11, T=13</td>
</tr>
<tr>
<td>Biquad Filters (l = 22)</td>
<td>*,+,-</td>
<td>*:2 cycle, +,-:1 cycle</td>
<td>*:2-stage pipelined, Functional Pipelining</td>
<td>*:2-stage pipelined, Structural Pipelining</td>
</tr>
<tr>
<td>Elliptical Filter (l = 34)</td>
<td>*,+</td>
<td>*:2 cycle, +,:1 cycle</td>
<td>*:2-stage pipelined, Multicycle operations</td>
<td>*:2-stage pipelined, Multicycle operations</td>
</tr>
</tbody>
</table>

### Table 3: Average number of iteration and frequency of convergence to optimal solution

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Number of features</th>
<th>Number of control steps</th>
<th>Optimum solution</th>
<th>Average iteration</th>
<th>Convergence frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACET</td>
<td>8</td>
<td>4</td>
<td>*:+,+,&lt; /,&amp;,V</td>
<td>1.05</td>
<td>100%</td>
</tr>
<tr>
<td>HAL</td>
<td>11</td>
<td>4</td>
<td>*:+,+,&lt; /,&amp;,V</td>
<td>4.19</td>
<td>100%</td>
</tr>
<tr>
<td>MAHA(b)</td>
<td>16</td>
<td>5</td>
<td>*:+,+,&lt; /,&amp;,V</td>
<td>5.15</td>
<td>100%</td>
</tr>
<tr>
<td>Band Pass Filter</td>
<td>29</td>
<td>7</td>
<td>*:+,+,&lt; /,&amp;,V</td>
<td>31.75</td>
<td>100%</td>
</tr>
<tr>
<td>Biquad Filters</td>
<td>22</td>
<td>8</td>
<td>*:+,+,&lt; /,&amp;,V</td>
<td>10.65</td>
<td>100%</td>
</tr>
<tr>
<td>Elliptical Filter</td>
<td>34</td>
<td>9</td>
<td>*:+,+,&lt; /,&amp;,V</td>
<td>7.33</td>
<td>100%</td>
</tr>
</tbody>
</table>

34b