Journal of Systems Architecture 57 (2011) 340–353

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Real-time scheduling on heterogeneous system-on-chip architectures using an optimised artificial neural network

Daniel Chillet *, Antoine Eiche, Sébastien Pillement, Olivier Sentieys

Université de Rennes 1, IRISA/INRIA, BP 80518, 6 Rue de Kerampont, F22305 Lannion, France

A R T I C L E I N F O

Article history:
Received 13 July 2009
Received in revised form 23 August 2010
Accepted 25 January 2011
Available online 12 March 2011

Keywords:
Heterogeneous multi-processor architecture
Task scheduling
Artificial neural network
Hopfield model
Inhibitor neurons

A B S T R A C T

Today's integrated circuit technologies allow the design of complete systems on a single chip which execute complex applications specified as a set of tasks. The tasks are managed by an Operating System whose main role consists in defining the resource allocation and the temporal scheduling. One of the main characteristics of these architectures is the heterogeneity of their execution resources which makes this scheduling complex.

In this paper, we propose a neural network based model for the design of heterogeneous multiprocessor architectures scheduler. Previous works have shown that neural networks using the Hopfield model can be defined to schedule tasks on an homogeneous architecture. This approach was extended to take platform heterogeneity into account. The work presented in this paper is based on a new neural network structure using inhibitor neurons. These neurons allow to limit the number of additional neurons and the number of network re-initialisations to reach convergence. We compare our network to the classic solutions based on the Hopfield neural network. These comparisons show that the number of neurons is reduced by a factor of more than two, which reduces the time convergence. We also compare our approach in terms of number of migrations with the PFair algorithm which is known as an optimal solution in the context of homogeneous architecture. The results show that our solution significantly limit the number of task migrations. Finally, we present results in the context of heterogeneous multiprocessor architectures, which is representative of complex system-on-chip.

1. Introduction

The increase in complexity of modern embedded applications has led integrated circuit designers to develop system-on-chip (SoC) architectures [1]. As shown in Fig. 1, an SoC includes a general purpose processor (GPP), specific blocks for specialised tasks (HW blocks), one or more specialised processors (i.e. digital signal processor, DSP) and, since few years, reconfigurable areas.

Based on this general organisation, these architectures now offer the capability to embed complete and complex systems. To efficiently execute applications and ease the execution management, designers propose to include Operating System (OS) in heterogeneous SoC [2]. These architectures can be compared to heterogeneous multi-processor systems for which the OS has to ensure scheduling of a set of N_T application tasks. Scheduling consists in defining the time interval $[t_j; t_j + C_i]$ (with C_i the Worst Case Execution Time, WCET, of task T_i) of the task execution and the supporting execution target R_k. In the context of heterogeneous platform, the scheduling is a non-resolved hard problem, for which an optimal solution is not guaranteed. Numerous works have been published on this issue, including the works of Cardeira [3] who showed that it is possible to define an artificial neural network (ANN) structure to solve the sheduling problem for homogeneous systems. The principal limitations of the proposed structure are the number of neurons necessary to model the problem, and the large number of re-initialisations required to ensure the network's convergence in a state representing a valid solution.

In this article, we propose a new structure of neural networks which provides a valid scheduling solutions on heterogeneous multiprocessor architectures, an important reduction in the overall number of neurons and a faster convergence towards a valid scheduling solution. Our main objective is to propose an efficient hardware implementation of the scheduler to support the required performances.

The remainder of the paper is organised as follows. In Section 2 we present the works relative to the scheduling problem and more particularly the use of neural networks. Section 3 presents our solution which relies on the use of inhibitor neurons. The network construction is explained and the convergence of such a network is presented. Section 4 presents a technique to optimise the number of neurons with the objective of reducing the convergence time. Section 5 illustrates our proposal using a number of experiments.

* Corresponding author.
E-mail address: Daniel.Chillet@irisa.fr (D. Chillet).

1383-7621/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2011.01.004

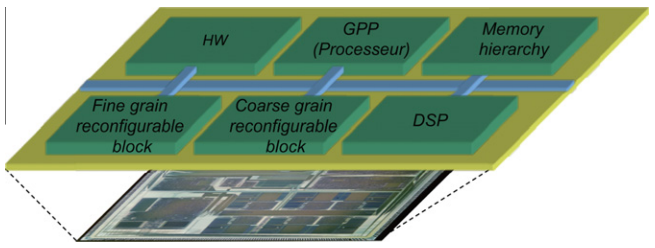
HW
GPP (Processeur)
Memory hierarchy
Fine grain reconfigurable block
Coarse grain reconfigurable block
DSP

Fig. 1. An SoC architecture example including a processor, a memory hierarchy, dedicated hardware accelerators and reconfigurable accelerators. Such systems can be seen as an heterogeneous multi-processor architecture.

We compare the results obtained using our solution with those obtained by Cardeira and we show the significant gain obtained especially in terms of network convergence. For the particular homogeneous multiprocessor context, we compare our proposal with the PFair algorithm and we show that our proposal provides less task preemptions and migrations. We also present some results of a typical application executed on a heterogeneous execution platform. A discussion about the complexity of our proposal is presented in Section 6. Finally, Section 7 discusses about the results and Section 8 concludes the paper and and presents some perspectives.

2. Modelling of the scheduling problem with ANN

Numerous proposals for the scheduling problem have been published over the years. For example, we can cite the rate monotonic [4], the deadline monotonic [5] or the earliest deadline first [6]. These approaches have been extended and there are numerous derivative propositions which allow to take particular constraints into account. The calculation of scheduling by these algorithms is carried out either before the execution of the tasks, i.e. off-line or statically [7], or at the execution time, i.e. in-line or dynamically [8,9]. In the context of SoCs, the system must react and adapt itself to external events, and such behaviour leds designers to favour on-line scheduling or mixed approaches [10]. However, these on-line approaches suffer from a high complexity which limits their use in real time systems.

Some works have concentrated on the problem of scheduling for multi-processor architectures. Algorithms producing optimal solutions have been proposed [11], as well as solutions sharing the tasks between multi-processor machines [12], but these studies consider homogeneous systems, i.e. systems with several identical processors. As a general rule, these solutions are not easily applicable in the context of SoCs due to the implicit heterogeneous nature of the architecture. The complexity of the scheduling problem as well as the time constraints (linked to the real time aspects of the applications running on the SoCs) have encouraged a number of works studying the implementation of OS services in hardware [13–15]. Among the numerous solutions to solve the optimisation problem, one uses artificial neural networks [3,16]. The underlying model is based on Hopfield's proposition [17]. In this model, each neuron n_i is connected to all other by a $W_{i,j}$ weight and receives an input energy I_i. The evolution of state x_i of neuron n_i is then given by:

$$x_i = \begin{cases} 1 & \text{if } I_i + \sum_{j=1}^{N} x_j \cdot W_{i,j} > 0 \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

The Hopfield's proposition defines an artificial neural network fully connected, capable of producing solutions to an optimisation problem by minimising an energy function. The definition of a Hopfield neural network goes traditionally through three steps which are:

- Model the problem in such a way that the neurons' states define a possible solution;
- Define the energy function that expresses a correct solution from the neurons' states;
- Compute the values of the weights of the connections $W_{i,j}$ between neurons and of the input energy I_i to each neuron.

From Hopfield's model, the authors of Ref. [3] suggest a modelling of the scheduling problem for homogeneous architectures. Their solution extends the results obtained in [18] where the theoretical basis for ANN design for optimisation problems are defined. By using a Hopfield model, they ensure the existence of a Lyapunov function, called *energy function* [19]. This model ensures that the network evolution converges towards stable states. Among these states, there is at least one solution which satisfies the constraints and produces an optimal solution. However, among these states, it is possible to obtain invalid solutions which correspond to local minima. The presence of these minima needs to add some energy to extract the network from these specific point and to ensure a new convergence. From the Hopfield model, the energy function must be written as

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} W_{i,j} \cdot x_i \cdot x_j - \sum_{i=1}^{N} I_i \cdot x_i \quad (2)$$

where N is the total number of neurons. To ensure the convergence, the neurons need to be randomly and sequentially evaluated. If this evaluation protocol is respected, it has been demonstrated that the evolution of the network will naturally lead to the minimisation of the energy function [20,21]. Thus, the network converges to a stable state, and the energy is minimum.

The application of this model to the task scheduling problem is shown in Fig. 2a where each circle represents a neuron, each line is the set of neurons allocated for one task T_l and each column is a scheduling cycle. The number of column N is equal to the total number of possible scheduling cycles. The state x_i (active or inactive) of the neuron n_i represents the state of the task (running or suspended). An active neuron ($x_i = 1$, black circles/neurons in Fig. 2b) indicates that the task must be placed in the running state and thus uses execution resource, while an inactive state ($x_i = 0$) represents a suspended task. The last line corresponds to a fictive task T_f which allows to model the inactivity of resources when no task needs to be executed. The computation load C_l of task T_l is defined as the number of time steps that the task needs to be completed on the execution resource, also known as the WCET of the task. Example of Fig. 2 corresponds to an application which has three tasks T_1, T_2, T_3, with WCETs $C_1 = 3, C_2 = 2, C_3 = 4$, periods $P_1 = P_2 = P_3 = 12$, and the total number of scheduling cycles $N = 12$. We also consider that the deadlines of a tasks are equal to their periods. Fig. 2b shows an example of a valid result of network convergence.

The proposed model relies heavily on the definition of a network construction rule, namely rule *k-OutOf-N*, allowing the specification that k neurons among N must be activated. This rule is defined by the energy function

$$E_{k\text{-}OutOf\text{-}N} = \left(k - \sum_{i=1}^{N} x_i\right)^2. \quad (3)$$

As presented in Fig. 3 for $k = 6$ and $N = 10$, this function is minimal and equal to zero when k neurons are active.

It has been shown in Ref. [3] that the neural network can be built by the addition of *k-OutOf-N* rules. The construction of the network is then ensured by applying this rule to a series of neuron groups. Fig. 4a and b shows the construction of a network for a mono-processor system for the same example as in Fig. 2. The

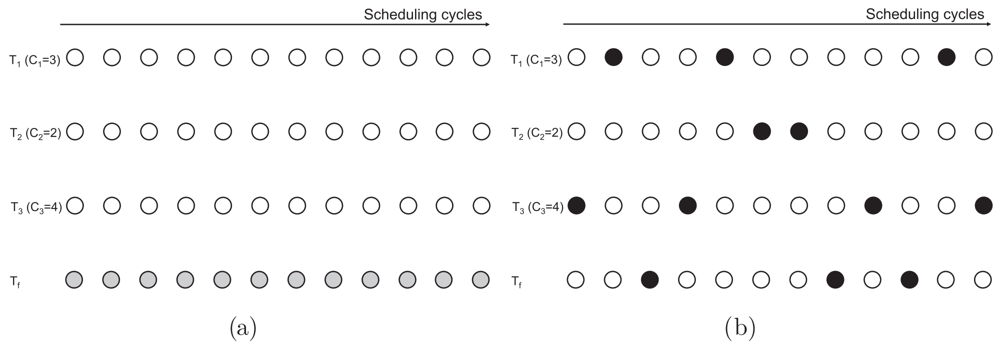

Fig. 2. (a) Initial representation of the scheduling problem of three tasks on a single execution resource. Each line corresponds to a task T_l with a WCET C_l, and the line T_f corresponds to a fictive task which allows to model the inactivity of resource when no task needs to be executed. (b) Convergence example of the neural network, the network state represents a valid solution.

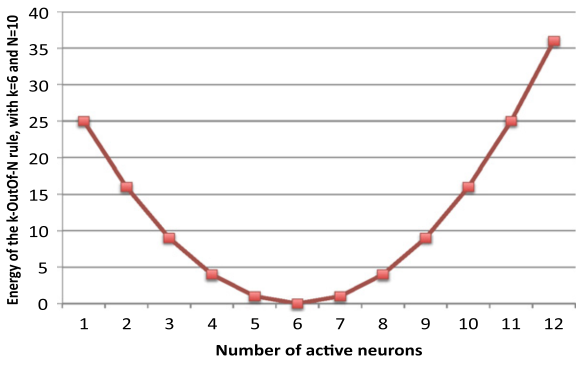

Fig. 3. Energy function evolution of a *k-OutOf-N* rule, with $k = 6$. This function has its minimum when six neurons are actived.

k-OutOf-N rule is first applied horizontally on each line of neurons for each task (see Fig. 4a), with $k = C_i$ for the application of the rule to task T_i, and N is equal to the total number of scheduling cycles ($N = 10$ in this example). The application of this rule makes sure that each task has the necessary number of active cycles to be executed. The *k-OutOf-N* rule is then applied to each column of the network in order to make sure that only one task is executed during each cycle (with $k = 1$ and $N = 4$, see Fig. 4b). This constraint is necessary to model a mono-processor with a mono-thread execution model.

Furthermore, if the execution of the group of tasks requires fewer cycles than the total number N, it is necessary to have a fictive task *Tf* which becomes active at the time steps where there is no more task to schedule. For this fictive task, a *k-OutOf-N* rule must be applied with $k = N - \sum_{i=1}^{N_T} C_i$, with N_T the number of application tasks.

Finally, the energy function needs to be rewritten as a Hopfield formulation. The *k-OutOf-N* construction rule (Eq. (3)), which is used to model the problem, is then rewritten to match the form of Eq. (2),

$$E_{k\text{-}OutOf\text{-}N} = -\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1\\ j\neq i}}^{N}(-2)\cdot x_i\cdot x_j - \sum_{i=1}^{N}(2k-1)\cdot x_i + k^2. \quad (4)$$

This allows the determination of the weights of the connections and the inputs to the neurons, which are given by

$$W_{i,j} = -2\cdot\overline{\delta_{i,j}} \quad \forall i = 1\ldots N, \forall j = 1\ldots N \quad (5)$$

$$I_i = 2k-1 \quad \forall i = 1\ldots N \quad (6)$$

with $\overline{\delta_{i,j}}$ the complement of Kronecker's symbol which takes the value 0 if $i = j$, and 1 otherwise. We can note that the connection and input values are independent of the k^2 term of Eq. (4) which corresponds to a constant offset. This term has no influence on the energy minimality.

Based on the *k-OutOf-N* rule and using the additive character of Hopfield's model, the scheduling problem for a single processor architecture is then easy to model. However, this simplicity hides

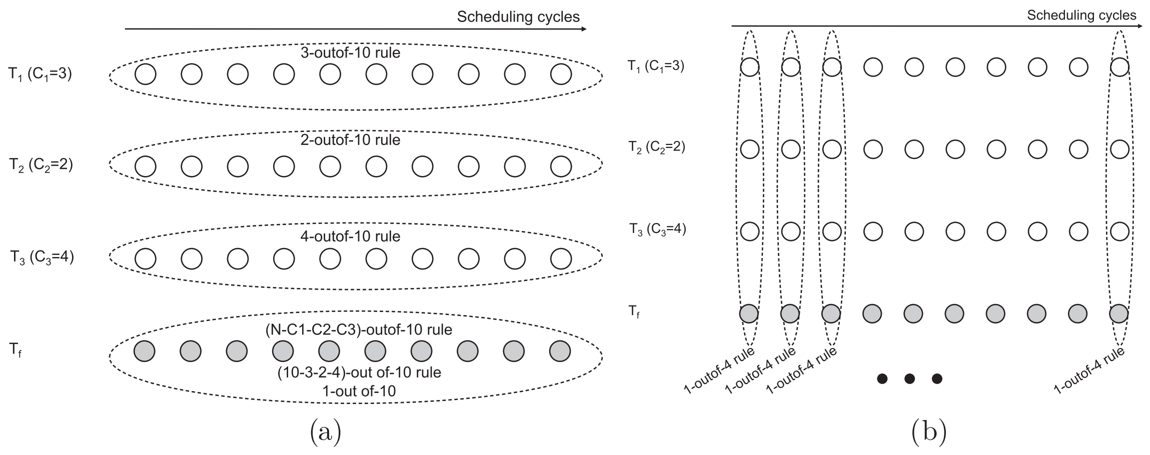

Fig. 4. (a) Application of *k-OutOf-N* rule on the lines of the network (with $k = C_l$ the WCET of task T_l on a single execution resource). For the last line (fictive task) the *k-OutOf-N* is applied with $k = N - \sum_{i=1}^{N_T} C_i$, with N_T the number of application tasks. (b) Application of *k-OutOf-N* rule on the columns of the network (with $k = 1$ in the case of a single processor executing a single task at each time).

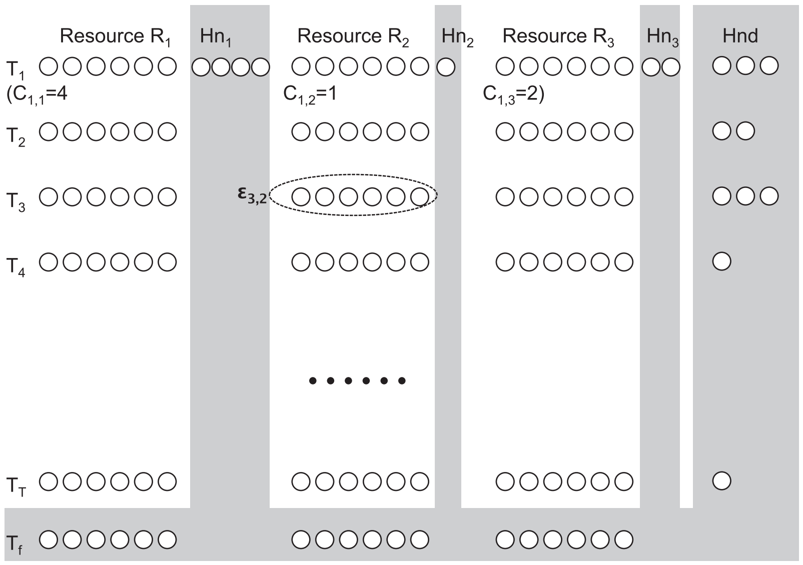

Fig. 5. Scheduling problem of T tasks on a heterogeneous architecture with three execution resources modelled by a neural network. The neurons placed in the grey zone, called Hn_i, ensure the convergence of (0-*or*-k)-*OutOf*-N rules applied on each execution target p. Neurons placed in the area called *Hnd* ensure the instanciation of task on a unique execution target.

a delicate problem of network convergence. As the authors of Ref. [3] have shown, the successive applications of several construction rules create several local minima in the energy function. These minima lead the network to converge in stable states which are not valid solutions for the problem to solve. In order to make the network converge towards a satisfactory solution, it is necessary to add energy to the network to extract from these minima. This requires a complex control mechanism. The main limitation of this proposal concerns the impossibility to schedule a multiprocessor architecture and this is clearly not satisfactory for a system made up of multiple execution resources, as is the case in today's SoCs.

The idea has therefore been extended for heterogeneous multiprocessor architectures [22]. The general idea consists in ensuring that if a task is to be executed once on a target type p, then it can not be scheduled on another target at the same time. Each execution target is represented by a plan p of neurons as in Fig. 5. To ensure that tasks are executed on only one execution target, a new rule is defined. This rule extends the k-*OutOf*-N rule towards a new one called (0-*or*-k)-*OutOf*-N. This rule allows us to define that the number of active neurons for a task T_l and for one specific execution target p is equal either to 0 or to $k = C_{l,p}$ (with $C_{l,p}$ the WCET of the task T_l on the execution target p). For that, the neural network is extended and specific neurons, i.e. hidden neurons, are added to ensure convergence. In Fig. 5, the hidden neurons are placed in the grey zones and are dependent on the WCET $C_{l,p}$. For S scheduling cycles (hyper-period) and R execution resources, we need to place a set of S neurons for each task and each execution resource. This set is called $\varepsilon_{i,j}$. For each set $\varepsilon_{i,j}, C_{i,j}$ hidden neurons (called $Hn_{i,j}$) are added in the hidden zone. For example, for task T_1 defined by the WCETs $\{C_{1,1}, C_{1,2}, C_{1,3}\} = \{4, 1, 2\}$, four neurons are added in the grey zone Hn_1 of execution resource R_1, see Fig. 5. For each task $T_i, max(|C_{i,j} - C_{i,k}|)\forall(j,k)$ neurons are added as global hidden neurons *Hnd*. Finally, S neurons are added for each execution resource to model the fictive task T_f. The number of useful neurons N_u (which represents the solution) for a set of T tasks is given by

$$N_u = R \cdot S \cdot (T + 1). \tag{7}$$

But the number of necessary neurons N_n to model the complete problem is given by

$$N_n = N_u + \underbrace{\sum_{i=1}^{T}\sum_{j=1}^{R} C_{i,j} + \sum_{i=1}^{T} max(|C_{i,j} - C_{i,k}|)\ \forall j,k = 1,2,\ldots R}_{\text{number of extra neurons}}. \tag{8}$$

As we can see in this expression, the cost due to extra neurons depends on the WCET of each task on each processor. These extra neurons are the hidden neurons needed to ensure a correct convergence. Eq. (8) shows that the number of hidden neurons changes linearly with the number of tasks but also depends on the characteristics of the tasks. For complex applications, the total number of neurons required to model the problem is then critical to ensure a rapid convergence and have a great impact on the implementation cost.

Concerning the network connectivity, the input and connection values must be defined by application of the following rules:

- a (0-*or*-k)-*OutOf*-N rule on the sets $\varepsilon_{i,j}$ of each execution resource, with $k = C_{i,j}$ and $N = S$;
- a k-*OutOf*-N rule on the sets $\varepsilon_{i,j} \bigcup Hn_{i,j}$ of each execution resource extended with the hidden neurons, with $k = C_{i,j}$ and $N = S + C_{i,j}$;
- a k-*OutOf*-N rule on the sets $\varepsilon_{i,1} \bigcup \varepsilon_{i,2} \bigcup \cdots \bigcup \varepsilon_{i,R}$, with $k = C_{i,j}$ and $N = R \cdot S$;
- a k-*OutOf*-N rule on each column of neurons to model the possible processor inactivity, with $k = 1$ and $N = T + 1$;

Note that the additivity of the Hopfield neural network model allows to apply several rules on the same neurons. But, as we have said before, this additivity creates one or several local minima.

The main limitations of this proposition are the large number of neurons required for the model and the great number of re-initialisations of the network to ensure its convergence towards a correct solution. In this context, the problem of the rule additivity is again present, which moreover, increases with the number of used rules. In fact, it is noteworthy that the greater the number of used rules is, the more the number of local minima increases and the less the probability of convergence towards a correct solution is. This last point is particularly delicate regarding the control of the network, since it has to be able to detect the stabilisation of the network, to reject non-valid solutions, to add energy and to re-start a network convergence.

3. A model of ANN with inhibitor neurons

In this section, after the presentation of the model of tasks considered in our work, we present our neural network structure and explain how this structure can converge towards valid solutions.

As in the previous works presented by Cardeira, the tasks considered in this paper are periodic and we suppose that the deadline of the tasks is equal to the period. To simplify the presentation, we also consider that the tasks are independent. The dependencies between tasks could be modelled, but this case is not considered in this article. We also suppose that the tasks can not suspend itself, but they can be preempted at each scheduling cycle.

Our proposition relies for its novelty on the reduction of the number of neurons as well as on the fact that the network convergence towards a correct solution is faster than the classical solutions. The general idea which has been exploited in previous works (a task executed once on a target resource cannot be executed on another target at the same time) is used in a different way through the use of new neurons, called inhibitor neurons. These new neurons have the role of *capturing* the firing of a task on a target resource and of preventing this same task from being launched on another target [23].

The principle of operation of an inhibitor neuron relies on a particular connection of one specific neuron (the inhibitor neuron) with the neurons to be inhibited. For the sake of clarity, we group the S neurons of a task T_i on the resource R_j as a set $\varepsilon_{i,j} = \{n_{i,j,1}, n_{i,j,2}, \ldots n_{i,j,S}\}$. The application has T tasks and the architecture has P types of execution resources. We need to introduce an inhibitor neuron $nh_{i,j}$ for each task T_i on each resource R_j, leading to a total of $P \times T$ inhibitor neurons.

The *k-OutOf-N* rule is applied to the group of neurons $\varepsilon_{i,j}$ with $k = C_{i,j}$. The inputs of inhibitor neurons $nh_{i,j}$ are fixed at values $Ih_{i,j} = -C_{i,j} + 1$. And, in connecting the neurons $n_{i,j,k}$ of the set $\varepsilon_{i,j}$ to the inhibitor neuron $nh_{i,j}$ with a weight equal to 1, as soon as the group $\varepsilon_{i,j}$ has $C_{i,j}$ active neurons the associated inhibitor neuron $nh_{i,j}$ can become active. The state $xh_{i,j}$ of the inhibitor neuron $nh_{i,j}$ is given by

$$xh_{i,j} = Ih_{i,j} + \sum_{k=1}^{N} x_{i,j,k} \cdot W_{(n_{i,j,k};nh_{i,j})} \tag{9}$$

with $x_{i,j,k}$ the state of neuron $n_{i,j,k}$ of the task T_i, $W_{(n_{i,j,k};nh_{i,j})}$ the weight of the connection between the neuron $n_{i,j,k}$ and the inhibitor neuron $nh_{i,j,k}$. Since these connections $W_{(n_{i,j,k};nh_{i,j})}$ are always equal to 1 (blue arrows in Fig. 6),[1] the value $xh_{i,j}$ can be written as

$$xh_{i,j} = -C_{i,j} + 1 + \sum_{k=1}^{N} x_{i,j,k}. \tag{10}$$

And due to the *k-OutOf-N* rule applied on the set $\varepsilon_{i,j}$ (with $k = C_{i,j}$), the value $xh_{i,j}$ can be set to value 1 only when $C_{i,j}$ neurons are actives, i.e. when the task T_i has obtained all its required scheduling cycles.

The activation of one specific inhibitor neuron $nh_{i,j}$ must then inhibit the activation of the neurons of the other execution resources. This is ensured by fixing the weights $W_{(nh_{i,j};n_{i,j,k})}$ between the inhibitor neuron $nh_{i,j}$ and the neurons of task T_i at a value which is sufficiently negative. Fig. 6 shows an example of a set $\varepsilon_{i,j}$ of neurons for task T_i, with the number of schedule cycle $S = 5$, and for one execution resource R_j. In Fig. 6, if the inhibitor neuron $nh_{i,j}$ has been activated by the execution of task T_i on the resource R_j, then the neurons of the other resources can no longer be activated since the weights $W_{(nh_{i,j};n_{i,l,k})}$ (between inhibitor $nh_{i,j}$ and neurons of $\varepsilon_{i,l} \forall l \neq j$, red arrows in Fig. 6) are strongly negative. To ensure this operation it is necessary to fix $W_{(nh_{i,j};n_{i,l,k})}$ at a value such that

$$W_{(nh_{i,j};n_{i,l,k})} \leqslant -I_{i,l} \quad \forall j \neq l. \tag{11}$$

In associating an inhibitor neuron with each execution resource, it is possible to ensure the convergence of the network towards a solution having only a single set $\varepsilon_{i,j}$ of activated neurons. The global model is then represented in Fig. 7 for P execution resources and T tasks. The neurons placed in the grey zones (NH_j) are the inhibitor neurons of the execution resource R_j. The complete definition of the neural network is formulated as follow:

- a (*0-or-k*)-*OutOf-N* rule on the sets $\varepsilon_{i,j}$ of each execution resource, with $k = C_{i,j}$ and $N = S$;
- a *k-OutOf-N* rule on the sets of inhibitors neurons $\{nh_{i,j}\}_{\forall j}$ for each task T_i, with $k = 1$ and $N = T$;
- the inhibitor neuron inputs are fixed at $Ih_{i,j} = -C_{i,j} + 1$;
- the inhibitor neuron connections are fixed at

$$W_{(nh_{i,j};n_{i,l,k})} \leqslant -I_{i,l} \quad \forall i,j,k,l | j \neq l,$$
$$W_{(n_{i,j,k};nh_{i,j})} = 1 \quad \forall i,j,k.$$

In comparison with the solution defined in [22] (see Fig. 5), the number of hidden neurons is fewer and moreover does not depend on the WCET of tasks on the execution target. So, in this case, the number of necessary neurons N'_n to model the problem is given by

$$N'_n = N_u + \underbrace{T \cdot R}_{\text{extra neuron number}}. \tag{12}$$

In comparison with Eq. 8, the number of extra neurons is fixed and equal to 1 for each task and for each execution target. This small number of extra cost neurons ensures a limited influence on the convergence time.

We must note that the number of neurons representing the solution is unchanged with our proposal. Indeed, our proposal focus on the number of extra neurons which can significantly increase when the WCET of tasks is great. For example, if two tasks have very different periods (T_1 and T_2 with periods equal to $P_1 = 5$ and $P_2 = 100$), the number of neurons representing the solution for each plan is equal to 100. But in our case, the number of extra neurons is equal to one for each task and each plan, while it depends from the WCET of tasks in the previous case. If the WCET of task T_2 is equal to 80 then the number of extra neurons for this task is equal to 80.

In contrast to previously proposed models, this proposal is based on a non symmetric connectivity, as we can see on the overall connections of the complete structure shown in Fig. 8. This asymmetry concerns all the connections with inhibitor neurons. In fact, the state of an inhibitor neuron $nh_{i,j}$ depends on the corresponding neurons of task T_i and on resource R_j. On the other hand, there is no connection from neuron $nh_{i,j}$ to the neurons of task T_i of the execution resource R_j. It is important to note this asymmetry since it brings into question the stability conditions defined by Hopfield (especially the symmetry of the connections matrix). However, as we shown below, convergence of the network is ensured. This can be explained and formulated as follow:

1. *Initial condition:* Initially, the system is in a state such that all the inhibitor neurons are inactive, and all the other neurons $n_{i,j,k}$ of the sets $\varepsilon_{i,j}$ can be initialised in a random state (active or inactive).

$$nh_{i,j} = 0 \quad \forall i,j, \tag{13}$$
$$n_{i,j,k} = \text{random (active, inactive)} \quad \forall i,j,k. \tag{14}$$

[1] For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.

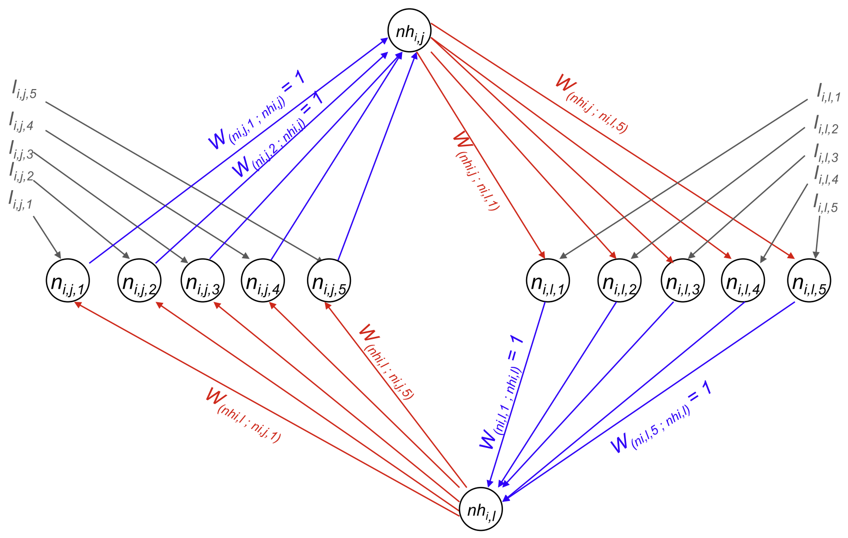
$nh_{i,j}$
$nh_{i,l}$
$I_{i,j,5}$
$I_{i,j,4}$
$I_{i,j,3}$
$I_{i,j,2}$
$I_{i,j,1}$
$I_{i,l,1}$
$I_{i,l,2}$
$I_{i,l,3}$
$I_{i,l,4}$
$I_{i,l,5}$
$n_{i,j,1}$ $n_{i,j,2}$ $n_{i,j,3}$ $n_{i,j,4}$ $n_{i,j,5}$
$n_{i,l,1}$ $n_{i,l,2}$ $n_{i,l,3}$ $n_{i,l,4}$ $n_{i,l,5}$
$W_{(ni,j,1\,:\,nhi,j)} = 1$
$W_{(ni,j,2\,:\,nhi,j)} = 1$
$W_{(nhi,j\,:\,ni,l,1)}$
$W_{(nhi,j\,:\,ni,l,5)}$
$W_{(nhi,l\,:\,ni,j,5)}$
$W_{(nhi,l\,:\,ni,j,1)}$
$W_{(ni,l,1\,:\,nhi,l)} = 1$
$W_{(ni,l,5\,:\,nhi,l)} = 1$

Fig. 6. Principle of inhibitor neurons utilisation for the task T_i, with five schedule cycles and two execution resources R_j and R_l. An inhibitor neuron $nh_{i,j}$ (respectively $nh_{i,l}$) is associated to the resource R_j (respectively R_l). The goal of the inhibitor neuron $nh_{i,j}$ consists in capturing the scheduling of the task T_i on the resource R_j and to inhibit the scheduling of this task on other resources $R_l \forall l \neq j$.

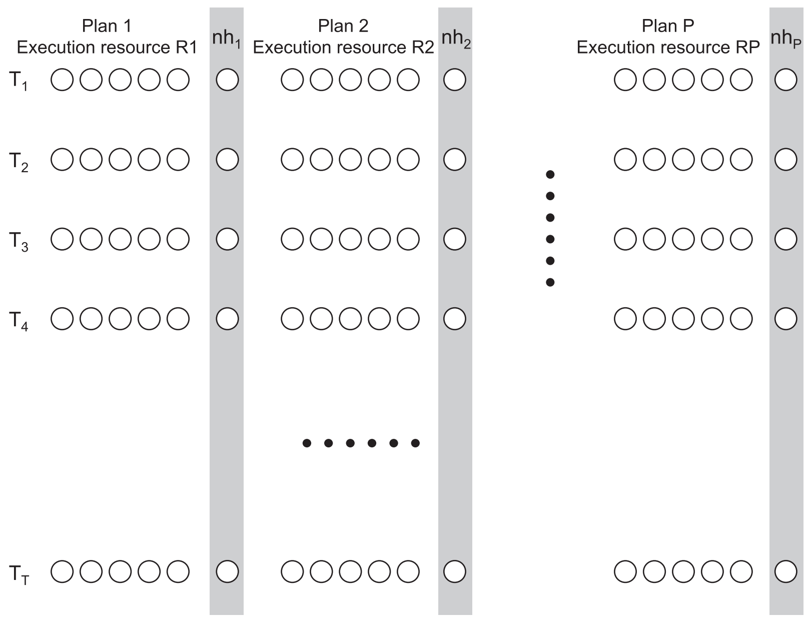

Fig. 7. Scheduling problem modelling with inhibitor neurons for P execution resources and T tasks. The inhibitor neurons are placed in the grey areas. This model needs only one inhibitor neuron by task and by execution resource.

2. *First convergence step:* From this initial state, each set $\varepsilon_{i,j}$ of neurons (corresponding to the execution of task T_i on resource R_j) will tend to converge towards a state such that $C_{i,j}$ neurons become active (due to the *k-OutOf-N* rule on neurons of resource R_j, with $k = C_{i,j}$).

$$\Delta(\varepsilon_{i,j}) \to C_{i,j} \quad \forall i,j \quad \text{while } nh_{i,j} = 0 \tag{15}$$

with $\Delta(\varepsilon_{i,j})$ the number of active neurons in the $\varepsilon_{i,j}$ set.

3. *Inhibition step:* When one of the plans R_j has $C_{i,j}$ active neurons (i.e. sufficient active neurons for the schedule of task T_i on resource R_j, $\Delta(\varepsilon_{i,j}) \geqslant C_{i,j}$), if the inhibitor neuron $nh_{i,j}$ is evaluated, then it becomes active. It is however the only combination which can lead to the activation of one of the inhibitor neurons.

$$nh_{i,j} = \begin{cases} 1 & \text{if } \Delta(\varepsilon_{i,j}) \geqslant C_{i,j} \text{ and if } nh_{i,j} \text{ is evaluated} \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

4. *Second convergence step:* When a single inhibitor neuron $nh_{i,j}$ is active for a task T_i and resource R_j, the weight of its connections with all the other neurons of the other execution resources $R_k \forall k \neq j$ (for the same task) cancels the weights of the inputs to these neurons. In this case, when the neurons of the other execution plans are evaluated, they will remain or become inactive.

$$\Delta(\varepsilon_{i,k}) \to \begin{cases} 0 & \text{if } nh_{i,j} = 1 \forall k | k \neq j \\ C_{i,k} & \text{if } nh_{i,k} = 1 \end{cases}. \tag{17}$$

The second convergence step leads the system to converge towards a state for what each task is executed on only one execution resource, with sufficient active scheduling cycles. The most important advantage of our proposal is that the convergence is always ensured towards a valid solution, which is not always the case for the classical methods.

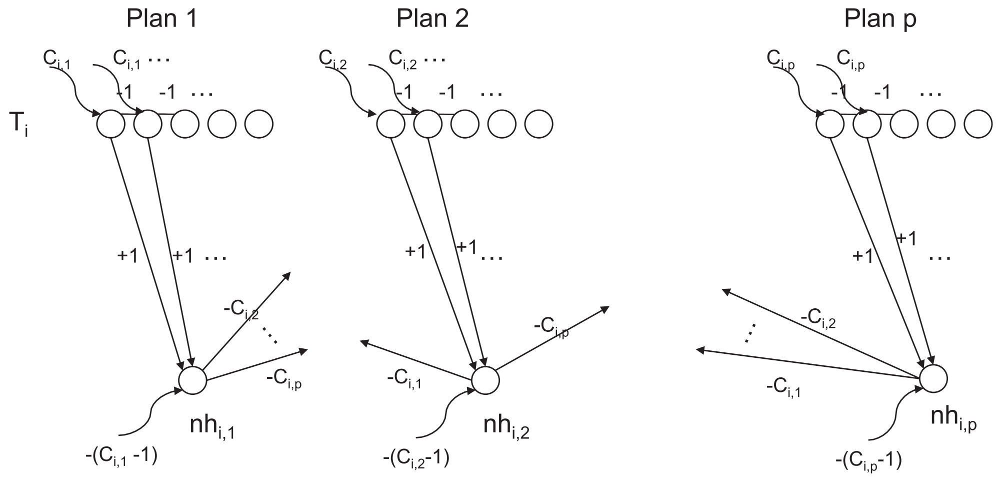

Fig. 8. Neuron connection for the scheduling problem using the inhibitor neurons.

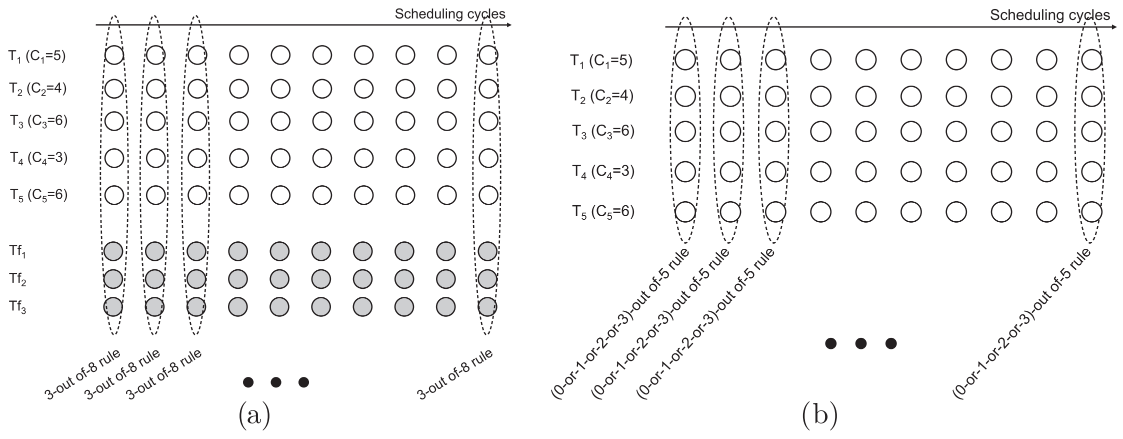

Fig. 9. Example of two neural networks and rule applications for three execution resources. (a) With the classical approach, three fictive tasks must be added to ensure the correct convergence of the network. (b) We propose to delete the fictive tasks, but in this case the rule to apply on each column must change to take account of the variable number of neurons that can be active on each column.

4. Optimisation of neural number

In the classical solutions proposed by Cardeira [3] (see Fig. 9a), β fictive tasks must be added in the neural network to model the inactivity of the β processors during some cycles. These tasks are necessary to ensure that exactly β neurons are active for each scheduling cycle (which corresponds to β tasks in the running mode for each cycle). Due to the additivity of the Hopfield model, the neuron input of the task T_i is equal to: $I_i = Ih_i + Iv_i$, with Ih_i and Iv_i respectively the inputs for the horizontal rule and for the vertical rule. The value Ih_i depends on the WCET of the task, and the value Iv_i depends on the number of tasks that can be placed simultaneously in the running mode (β). Fig. 9a presents an example of a neural network for a set of five tasks to schedule on three execution resources. In this case, three fictive tasks must be added and a *k-OutOf-N* rule is applied on each column of neurons, with $k = \beta = 3$ and $N = 8$. In order to limit the number of neurons, we propose to remove the neurons of fictive tasks, as presented in Fig. 9b. In this case, we need to apply a (0-*or*-1-*or*-2-*or*-3)-*OutOf*-5 rule on each column.

With a connection between vertical neurons j equal to $-2/\beta$ (for each cycle), the corresponding behaviour is obtained, with $\beta$the maximum number of tasks that can be executed in parallel at each cycle ($\beta = 3$ in the case of Fig. 10). The principle consists in removing a little energy for a neuron $n_{i,j}$ when the maximum number of tasks is scheduled at the cycle j. The removed energy is equal to -2 and is sufficient to prevent the *k-OutOf-N* horizontal rule to active the neuron $n_{i,j}$.

For example, let us suppose that the neural network presented in Fig. 9b is in the state presented in Fig. 10 (black neurons are active, white neurons are inactive and grey neurons are considered as inactive first). At this step, if the third neuron of task T_3 is evaluated, the *k-OutOf-N* rule applied in the neuron line of task T_3 leads to active this neuron (with $k = 6$ and $N = 10$). But the third cycle already has three tasks scheduled, so we need to prevent the task T_3 to be scheduled at this cycle. This is ensured through the specific connection of the neurons in column, equal to $-2/\beta$. Now, if the last neuron of task T_3 is evaluated, for the same reason, the *k-OutOf-N* rule applied on the neurons of task T_3 leads to active this neuron. In this case, because this last cycle does not have three tasks scheduled, it is possible to active the neuron of task T_3.

The results presented in Section 5.2 show that this construction allows to obtain a substantial reduction of the number of neurons with a fast convergence. The reduction of the number of neurons is equal to $\beta \times S$ (with S the scheduling cycles, i.e. the hyper-period). In the case of Fig. 9a and b, the reduction is equal to 30 neurons, which represents 27% of reduction.

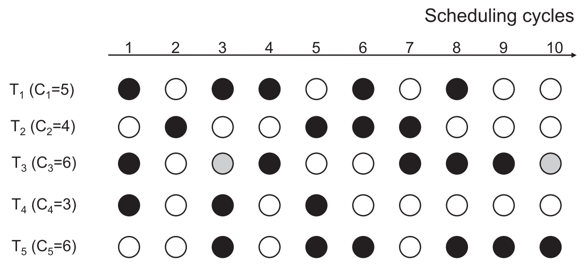

Fig. 10. Example of evolution of the neural network presented in Fig. 9b with three execution resources and thus three tasks that can be executed in parallel ($\beta = 3$). We suppose that tasks T_1, T_2, T_4, T_5 have obtained their schedule cycles and T_3 received only five of its six schedule cycles. The gray neurons are the next two neurons evaluated in the network evolution. The third cycle of task T_3 can not become active due to the three tasks already scheduled at this cycle (T_1, T_4 and T_5). The last cycle of task T_3 can become active due to the only one scheduled task at this cycle (T_5).

5. Results

In this section, we present comparisons of the proposed ANN structure with the previous solutions proposed in [16,22]. We show that our proposal is more efficient (i.e. convergence speed-up), reduces significantly the number of neurons, and limits the network re-initialisations. Although we have developed a solution for heterogeneous architecture, our proposal is able to schedule tasks onto homogeneous platform system. We also give some results to compare our approach with the optimal PFair algorithm in the homogeneous context. However, in the case of heterogeneous multiprocessor architectures where PFair is not applicable, we show that our solution is able to produce scheduling results.

5.1. Validation tools

To validate our proposal we have developed a neural network simulator, `SimulANN`. This tool, developed in Java, allows to define any neural network (i.e. number of neurons; neuron energy inputs and weight connections between neurons) and is coupled with a front end tool which facilitates the classical rule applications on neurons set, for example a *k-OutOf-N* rule on a set $\varepsilon_{i,j}$ of neurons. The starting point of the `SimulANN` tool is the specification of the application tasks characteristics. These characteristics concern the number of tasks, the number of execution resources, the number of schedule cycles, and the WCET of each task on each execution resource. From these information, the tool allows to simulate the neural network evolution. Some directives can be added to test the network convergence and stop when a valid solution is found.

5.2. Comparison with classical neural network approaches

The first example concerns the schedule of three tasks T_1, T_2 and T_3 on one execution resource. To solve this problem, the model of Ref. [16] needs to add one neuron line to model the processor idle cycles. In Fig. 11a and b the initial state of the network and one possible network final state are presented. The authors indicate that convergence is obtained after more than 600 fired neurons. Fig. 11c presents our model for the same scheduling problem. As we can see, the number of neurons is lower than for the classical solution. Indeed, in this model, the fictive tasks are deleted by modifying the *k-OutOf-N* vertical rule applied on each column of the neural network. Moreover, for all simulations and from a random initial state, our network always converges to a valid solution. This convergence is obtained with an average number of 79 fired neurons.

Fig. 12 shows the evolution of the energy function for one simulation. This simulation converges to a valid solution for only 65 fired neurons. Contrary to the classical energy function evolution, we can observe that this function decreases monotonously with the number of fired neurons.

Fig. 13a presents another example with two homogeneous execution resources which is modelled by the classical network. Fig. 13b shows that by applying the same modified *k-OutOf-N* vertical rule on each column of the neural network, the required neurons to model the same problem with regards to the classical proposition can be reduced. Moreover, the classical solution converges to a valid solution with more than 300 fired neurons, while our proposition limits the number of fired neurons to approximatively 70. The number of removed hidden neurons is directly proportional to the hyperperiod of the tasks, which can be important. This reduction will favourably impact the hardware implementation of the scheduler.

5.3. Comparison with PFair algorithm

In this section, we compare our scheduling with the PFair algorithm in the context of homogeneous multiprocessor architecture. The main drawback of the PFair algorithm is the large number of task preemptions and migrations required to ensure the complete

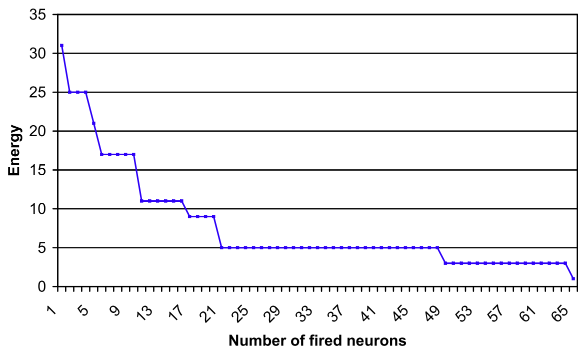

Fig. 12. Energy function evolution with our network model for the example of Fig. 11.

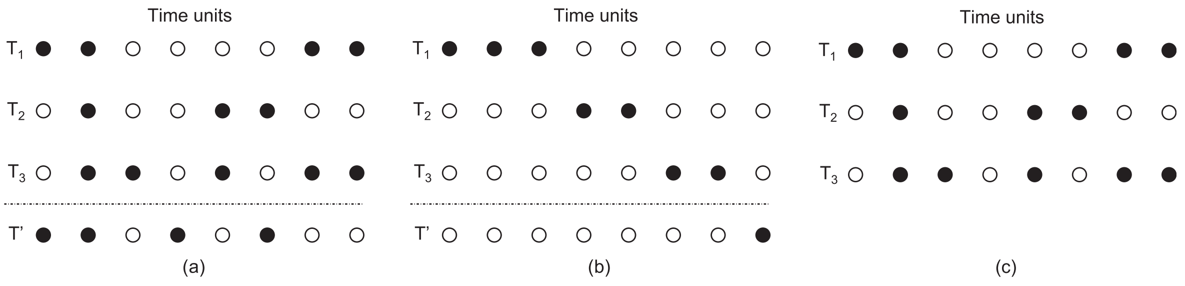

Fig. 11. Example of state representation of neural network (task WCET are equal to 3, 2, 2 for respectively T_1, T_2, T_3) (a) Initial state with one hidden neuron line. (b) Stable state and valid solution. (c) Initial state of our proposal without hidden neurons.

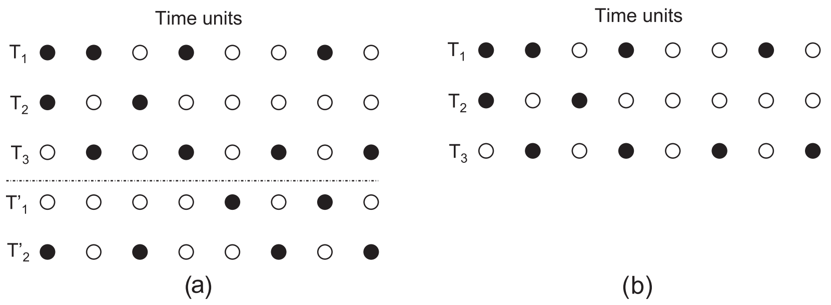

Fig. 13. (a) Initial state of the classical neural network. (b) Initial state of our neural network.

task scheduling. Conversely, our proposal is based on a strong constraint which ensures that tasks are always scheduled on one execution resource. This contraint can lead to an impossility of scheduling all the tasks, but when the schedule is possible, the number of migrations is nul.

The results presented in this section show that our proposal produces solution without any task migration.

For these comparisons, we use the set of independent tasks $\{T_i, \mathrm{WCET}_i\}$ defined as

$$\{T_i, WCET_i\} = \{(T_0,4),(T_1,5),(T_2,3),(T_3,7),(T_4,9),(T_5,6),(T_6,8),(T_7,4)\}. \quad (18)$$

A multiprocessor system composed of four processors is considered. The schedule period is equal to 20 and each task must be executed once on this period. This first simple example presents a global processors (or system) workload equal to 58%.

Fig. 14a shows the result of the task scheduling with the PFair algorithm, while Fig. 14b shows one result provided by our neural network. As we can see, the two schedulers produce a valid schedule solution where all the tasks are executed during the 20 schedule cycles.

Fig. 14a shows that the PFair algorithm generates a large number of task migrations between the four different processors. For example, task T_4 is scheduled on processor R_1 at cycle t_0, then scheduled on processor R_2 at cycle t_5, then scheduled on processor R_1 at cycles t_{11}. The total number of task migrations for the PFair scheduling is equal to 16 (0 for T_1, 1 for T_2, 1 for T_3, 2 for T_4, 2 for T_5, 5 for T_6, 5 for T_7, 0 for T_8). Each task migration consists in a task context saving on one processor and in a task context loading on another processor. These task context transfers are time consuming but completely ignored by the PFair algorithm. For a distributed memory system, the task context must be transfered between the processors for each migration. The same remark can be made for all multiprocessor systems with a level-1 instruction cache. Indeed, in this case this level-1 cache is always distributed. The time overhead of these transfers is a main drawback for the implementation of the PFair algorithm. Extension of PFair algorithms have been proposed [24] to limit the number of migrations, but some of them remain necessary to ensure the complete task scheduling.

As we can see in Fig. 14b, no task migration between the different processors is generated with our proposal. This characteristic is ensured by the inhibitor neurons which prevent the schedule of the same task on several processors. This characteristic allows to limit the memory accesses for the task context switches, and therefore removes the time overhead presents in the PFair algorithm.

Concerning the number of task preemptions, the two proposals generate preemptions but our neural network limits their numbers. Indeed, as we can see in Fig. 14a, the PFair scheduling generates 35 task preemptions, while our proposal generates only 21 task preemptions. The reduction of number of task preemptions

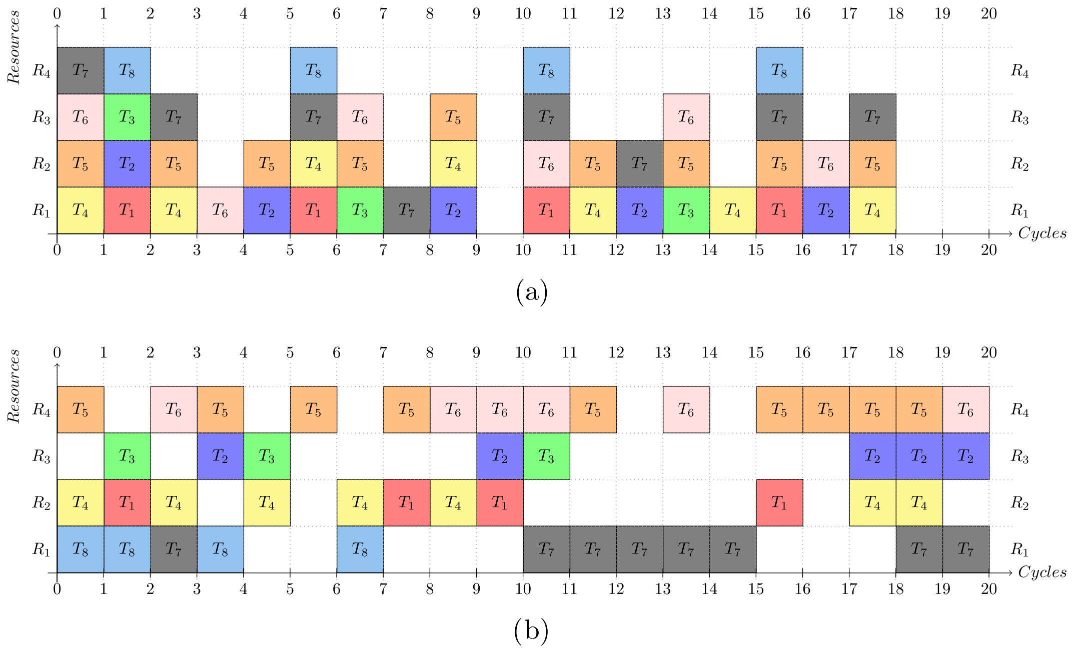

Fig. 14. Gantt diagrams for the scheduling of 8 tasks on a 4 resource multiprocessor system during 20 schedule cycles. (a) PFair scheduler. (b) Our neural network scheduler.

is also an important characteristic of our proposal. Indeed, each task preemption needs a context switch which is time consuming: the context of the current task must be saved and the context of the next task must be loaded.

We can note that it is also possible to compact all the scheduling cycles of each resource toward the left. For our proposal, the compacting phase is very simple due to the absence of task migration between processors. Fig. 15 shows how the solution generated by the neural network can be optimised. With the PFair algorithm, this compacting phase is not so simple because of the schedule of each sub-task on the different resources. The main problem in this case is the possibility to invert the order of sub-task and the risk to schedule two different sub-tasks of the same task at the same schedule cycle. For example, if we compact the PFair solution given in Fig. 14a, the first sub-task of T_5 on resource R_3 would be executed in parallel with the fourth sub-task of T_5 on resource R_2. For the solutions generated by our proposal, the compacting phase can be done without problem of sub-task order or parallel sub-tasks.

For the following results, we use the task generator TGFF [25] to generate large set of independent tasks. The number of tasks generated is equal to 30, the period for the scheduling is equal to 100 and the number of processors is equal to 5. The task executions are randomly fired in the interval [5; 15]. To try to limit the task migrations produced by the PFair algorithm, we apply the heuristic H2

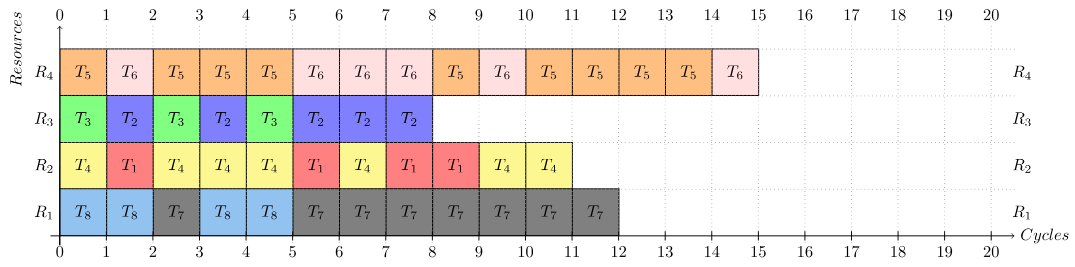

Fig. 15. Gantt diagrams for the schedule of 8 tasks on a 4-processor system during 20 schedule cycles after a compacting phase of task within the execution resources.

Table 1
Comparisons between PFair scheduler and our neural network scheduler.

Graph number	Workload (%)	PFair scheduler with H2 heuristic			Neural network scheduler		
		Valid schedule	Number of preemptions	Number of migrations	Number of tasks scheduled (%)	Number of preemptions	Number of migrations
G_1	75.0	100%	308	146	95.9	338.7	0
G_2	74.5		306	142	97.7	336.5	
G_3	71.3		290	115	99.5	330.8	
G_4	76.8		315	86	87.9	345.5	
G_5	77.3		317	110	92.1	343.4	
G_6	68.0		278	82	100.0	322.1	
G_7	73.8		299	85	96.8	335.9	
G_8	76.8		315	114	90.7	343.7	
G_9	81.0		331	120	64.5	355.8	
G_{10}	71.3		293	115	97.1	329.9	
G_{11}	82.0		335	109	64.5	358.3	
G_{12}	65.3		266	42	100.0	318.1	
G_{13}	72.3		297	101	98.6	331.3	
G_{14}	75.8		311	86	91.1	341.5	
G_{15}	72.8		297	90	98.9	334.5	
G_{16}	67.3		277	98	100.0	322.7	
G_{17}	75.3		309	97	98.1	336.8	
G_{18}	78.0		320	113	83.7	348.4	
G_{19}	77.3		317	113	85.8	345.9	
G_{20}	75.5		310	94	59.5	341.2	
G_{21}	74.0		304	88	96.9	334.5	
G_{22}	70.3		288	94	100.0	327.4	
G_{23}	73.8		303	99	96.5	337.3	
G_{24}	78.8		321	85	80.5	350.1	
G_{25}	69.0		284	96	99.7	325.5	
G_{26}	73.8		303	92	98.2	334.3	
G_{27}	74.0		304	99	95.4	338.1	
G_{28}	67.0		272	112	100.0	320.6	
G_{29}	73.8		300	94	97.9	336.3	
G_{30}	79.3		323	95	73.1	352.4	
G_{31}	80.0		328	86	73.4	353.5	
G_{32}	78.3		319	85	79.6	350.1	
G_{33}	75.3		306	113	90.1	341.5	
G_{34}	79.3		325	101	81.6	349.7	
G_{35}	76.8		315	100	89.5	343.8	
G_{36}	75.3		307	123	95.9	336.8	
G_{37}	78.5		322	71	80.6	348.7	
G_{38}	76.0		308	103	90.7	341.1	
G_{39}	78.0		320	143	86.2	347.4	
G_{40}	74.0		303	90	94.9	336.82	
Average	74.8	100%	306.2	100.7	90.95	339.6	0

presented in [24]. Table 1 compares the results for a set of 40 task graphs (G_i is the graph name). The first three columns give the PFair scheduler performances: firstly the capability of the PFair to schedule the task graph, secondly the number of task preemptions, and thirdly the number of task migrations. The other columns give the performances of our proposal for the same three parameters.

Firstly, we can observe that the PFair scheduler always obtains a solution. For all these produced solutions, the average number of migrations is about 100, and the average number of preemptions is about 306. As we explain before, our scheduler does not obtain a complete solution for each convergence. For the set of graphs presented here, the average number of scheduled tasks is 91%. For each incomplete schedule, a small number of neural network re-initialisations is sufficient to converge towards a complete solution, see Section 5.4. Concerning the number of preemptions, our proposal produces approximately 10% more than the PFair algorithm, but the major advantage of our proposal is the complete cancellation of migrations. As we have said before, this last point is a major drawback of the PFair algorithm, and although our proposal does not target homogeneous multiprocessor, it can provide interesting alternative for this context.

Table 2
a) Example of task characteristics for two types of execution resources. b) Input file of the `SimulANN` simulation tool for neural networks.

Tasks	$C_{i,1}$	$C_{i,2}$
(a)		
T_1	1	2
T_2	2	1
T_3	4	2
T_4	3	5
T_5	4	6
T_6	3	2
T_7	2	3

(b)
```
Tasks T1 T2 T3 T4 T5 T6 T7
Plans P1 P2
NbProcByPlans 1 1
SchedulingInterval 20
WCETByPlan T1 1 2
WCETByPlan T2 2 1
WCETByPlan T3 4 2
WCETByPlan T4 3 5
WCETByPlan T5 4 6
WCETByPlan T6 3 2
WCETByPlan T7 2 3
```

5.4. Results on heterogeneous architectures

We have applied the proposed model and the model presented in [22] to an architecture composed of two different types of resources, each type having one execution target available. Table 2a gives the list of tasks with the WCET of each task on the two execution resources. Table 2b is the input file of the `SimulANN` tool. The results of the simulations on a group of tasks varying from 2 to 7 are given in Table 3 for the classical model, and in Table 4 for our proposal.

For the classical model, the number of network re-initialisations corresponds to the number of simulations necessary to ensure the convergence towards a valid solution. For each invalid solution produced, we need to restart the system. For all the performed simulations, we noted that approximately 10 neurons are evaluated before stopping the convergence and restart it.

Results in Table 3 show clearly that the total number of neurons needed to model the problem with the previous approach is significantly higher than the useful neurons. The extra cost line of the table represents the ratio between the total number of neurons N_n and the number of useful neurons N_u, see Eq. (8). Similarly, this table shows that the model requires a number of re-initialisations of the network which makes its practical use impossible in real-time in the context of SoCs, where the aim is to calculate the scheduling in-line.

The scheduling results for the proposed structure of neural network with the concept of inhibitor neurons are given in Table 4. These results show that the percentage of extra neurons needed to model the problem remains low and does not increase with the number of tasks for a given number of execution cycles. We can note that in the previous proposals, especially in [3], a large number of hidden neurons are added to the model, which leads to an increasing convergence time. In our case, the extra costs incurred by the inhibitor neurons is 5%, which remains acceptable. The reduction of the total number of neurons is between a factor of 1.6 and a factor of 2.1 compared to the previous works. Furthermore, this extra cost does not depend on the number of scheduling cycles, which is a huge advantage for real-life applications.

Table 3
Results for the classical neural network model for the scheduling of 2–7 tasks, for an architecture having one execution resource of two different types, and for 20 scheduling cycles. Note that, in average, 10 neuron evaluations are necessary to obtain a stable network state, but many re-initialisations are necessary to converge towards a valid solution.

Number of tasks	2	3	4	5	6	7
Number of neurons (N_n)	180	240	306	364	416	468
Number of useful neurons (N_u)	80	120	160	200	240	280
Extra cost of hidden neurons	2.25	2	1.91	1.82	1.73	1.67
Average number of network re-initialisations	54	338	1492	9191	31783	28546
Average number of neural evaluations	545	3379	14920	91910	317830	285460

Table 4
Results for the proposed neural network model with inhibitor neurons for the scheduling of 2–7 tasks. The architecture has two plans each with one execution resource, and for 20 scheduling cycles. The three last lines highlight the reduction of our proposal in terms of total number of neuron evaluations. Because the average number of re-initlisations is too small, we present the maximum value found in the total neural network simulations.

Number of tasks	2	3	4	5	6	7
Number of neurons	84	126	168	210	252	294
Number of useful neurons	80	120	160	200	240	280
Extra cost of hidden neurons				1.05 (+5%)		
Neuron reduction	2.1	1.9	1.8	1.7	1.65	1.6
Max number of network re-initialisations	0	0	0	1	2	5
Average number of neural evaluations	339	539	1025	1170	1583	1951
Gain in number of evaluations	1.6	6.3	14.5	78	200	146

If the hardware cost of the implementation is an important issue, convergence time is also of great interest. On this point, our solution has a much more significant improvement. The three last lines of Table 4 show the convergence efficiency of the proposed approach compared to classic solutions. We can notice that our proposal greatly reduces the number of network evaluations (up to a factor of 200) and the number of network re-initilisations. Moreover, we can note that this benefit will increase with the application complexity. This is an important characteristic which reinforces the idea that this type of structure is particularly well suited for an SoC platform.

Concerning the scheduling for seven tasks, an example of result is presented in Fig. 16. We can notice that the schedule of each task is always exclusive, i.e. if a task is scheduled on resource i, this task is never scheduled on resource j with $i \neq j$. For example, tasks 2, 3, and 6 are scheduled on the execution resource type 1, and tasks 1, 4, 5, and 7 are scheduled on the execution resource type 2. The figure also shows the inhibitor neurons of each execution plan (in the grey zones).

5.5. Results in the context of a system-on-chip architecture

Modern SoC architectures are composed of several heterogeneous resources (typically between five and ten resources) and execute applications which are composed of several tasks (typically ten to twenty). In the general case, several tasks are defined for different resources, while the others are defined for one specific resource. This section shows the results obtained for a specific architecture which is composed of five different resources: resource R_1 is a GPP, resources R_2, R_3 and R_4 are specialised intellectual property (IP) blocks and resource R_5 is a dynamically reconfigurable accelerator (DRA). The complete application is composed of 10 tasks and we suppose that these tasks must be scheduled in 10 cycles. Seven tasks have been described for the GPP resource $(T_2, T_3, T_4, T_6, T_8, T_9, T_{10})$. Three tasks have been defined for the three IP blocks (T_1, T_5, T_7). Finally, six tasks have been also described for DRA $(T_2, T_3, T_6, T_8, T_9, T_{10})$. These information are summarised in the matrix C (Eq. 19), which defines the WCET of each task T_i on each resource R_j (∞ indicates that the task cannot be executed on the corresponding resource).

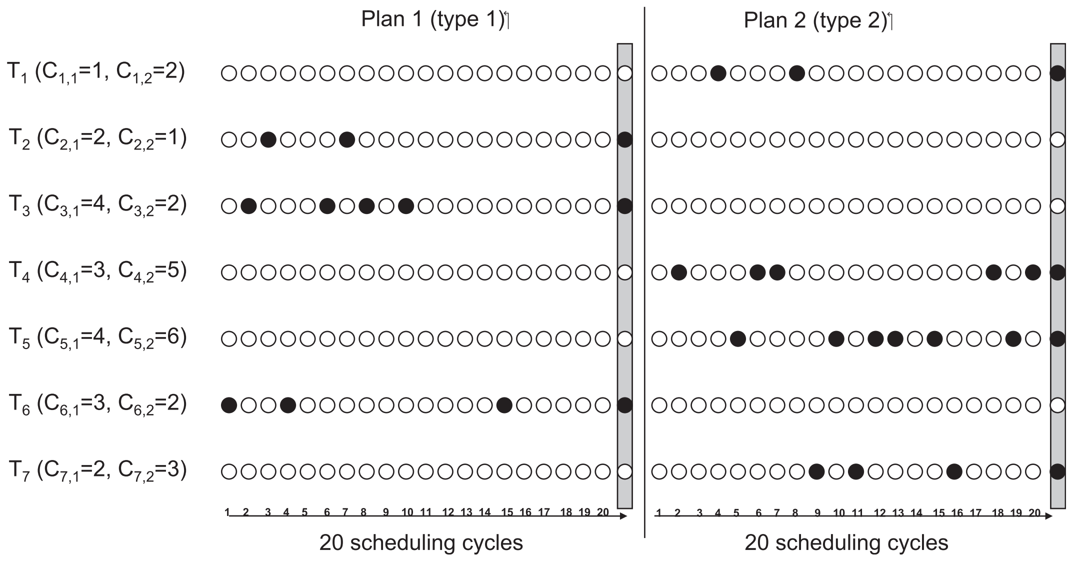

Fig. 16. Scheduling example of the proposed neural network model for an application with seven tasks, for an architecture with two plans each with one execution resource, and for 20 scheduling cycles. Each task is scheduled only one time on one of the two execution plans. If a task is executed on an execution plan the corresponding inhibitor neuron is then active.

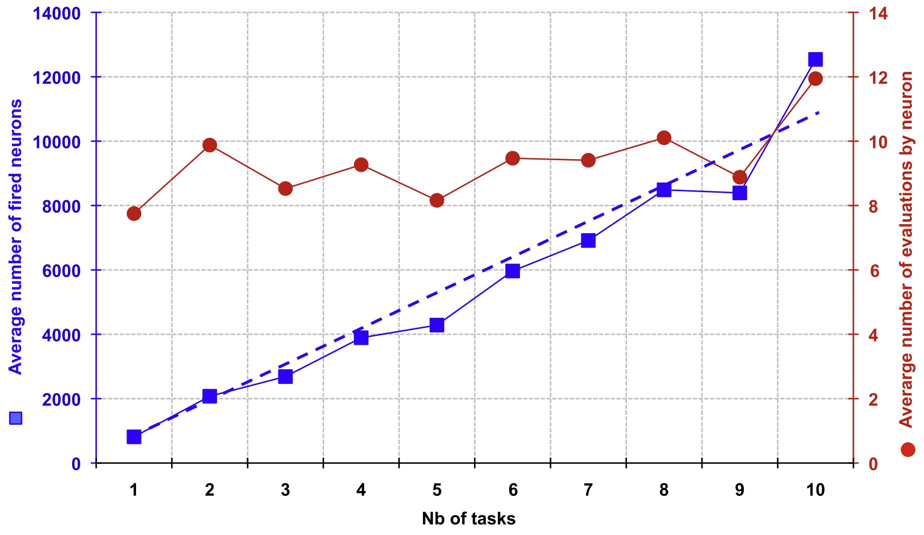

Fig. 17. Results for a SoC architecture running an application. This application is composed of ten tasks, which are introduced step by step in the scheduling neural network.

$$C = \{C_{i,j}\} = \begin{array}{|cccccccccc|c} \infty & 2 & 2 & 4 & \infty & 4 & \infty & 4 & 4 & 2 & R_1 \\ \infty & \infty & \infty & \infty & \infty & \infty & 10 & \infty & \infty & \infty & R_2 \\ \infty & \infty & \infty & \infty & 5 & \infty & \infty & \infty & \infty & \infty & R_3 \\ 4 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & R_4 \\ \infty & 2 & 1 & \infty & \infty & 2 & \infty & 2 & 1 & 2 & R_5 \\ T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & T_8 & T_9 & T_{10} & \end{array}. \quad (19)$$

Fig. 17 presents the results obtained on the application with the number of tasks varying from 1 to 10. The complete model of this problem requires 10 neurons per task and per resource. So 500 neurons are necessary for the complete neural network. Nevertheless, because some tasks cannot be scheduled on all resources, the network can be reduced and some neurons can be deleted from the model. For example, task T_1 can only be scheduled on resource R_4 so it is not necessary to place neurons for its schedule on the other resources. On the other side, because task T_2 is executable by resources R_1 and R_5, we need to place 2×10 neurons for its schedule.

Finally, only 160 neurons are necessary to model this problem. Fig. 17 shows that the number of fired neurons evolves linearly with the number of tasks, i.e. linearly with the neural network complexity. Another important result is the relatively constant number of evaluations of each neuron. In our example, each neuron is evaluated between 8 and 12 times. These results show that the time convergence of our proposal remains controlled. This is mainly due to the limitation of neurons number which ensures a fast convergence without re-initialization.

6. Complexity

Event if our contribution does not target a software implementation, we can briefly analyse the complexity of our neural network in comparison with to the previous solutions.

For each neuron evaluation, the calculation to achieve is gievn in Eq. (1). In this equation, the sum is done with each neuron connection. In our neural network, each neuron $n_{i,j,k}$ representing the scheduling cycle k for the task i and for execution resource j needs to be connected to all other neurons of the same task and for the same execution resource ($n_{i,j,m} \forall m \neq k$). Furthermore, the neuron $n_{i,j,k}$ of scheduling cycle k needs to be connected to all other neurons of the same scheduling cycle ($n_{m,j,k} \forall m \neq i$). Finally, the neuron $n_{i,j,k}$ needs to be connected to all inhibitor neurons of all other execution resources ($nh_{i,m}\ \forall m \neq j$) The number of multiplications and additions are then equal to

$$Nb_{mult} = Nb_{add} = S + T + R \quad (20)$$

With S the scheduling cycles, T the number of tasks and R the number of execution resources.

For the solution presented in [22], each neuron of a specific task and a specific execution resource needs to be connected to each other neurons of the same task for each execution resource. In this case, the number of multiplications and additions are then equal to

$$Nb'_{mult} = Nb'_{add} = S * R + T. \quad (21)$$

As we can see, our proposal limits the number of connections between neurons, and this limitation could be interesting for a software implementation of the our neural network. However, as mentionned in the introduction, our objective is to target an efficient hardware implementation of the neural network, in particular by exploiting the parallel evolution of the neurons in the network. In this context, the software complexity is an interesting information, but does not represent the hardware implementation complexity. We are currently working on the hardware implementation of the network in order to optimised the performances, but this part is out of the scope this article.

7. Discussions

Several studies have been proposed for multiprocessor scheduling and can be divided into two main techniques. The first is the *partionning* technique for which each task is firstly assigned to a processor and then scheduled on the processor. The second is the *global scheduling* technique for which tasks are managed globally and can migrate from a processor to another one. The most efficient algorithm is a global scheduling technique based on *proportionate fair* scheduling (PFair). This algorithm is known as an optimal scheduling for periodic and/or sporadic set of tasks. The complexity of this algorithm is polynomial and depends on the number of tasks to shedule. Due to its complexity, the PFair algorithm is generally not implemented in multiprocessor systems, and simpler schedule algorithms, such as EDF (earliest deadline first), are generally used. Compared with this solution, the number of neurons of our proposal remains linear for a given number of cycles, as shown in Fig. 17. Indeed, the convergence of the neural network depends on the number of neurons, and, in our context, this number is defined by the number of tasks to schedule, the total number of execution resources and the number of cycles of the scheduling period. In Fig. 17, the dotted line shows that, for a given number of cycles and a given number of execution resources, the convergence time is approximatively linear. Due to the inhibitor neuron behaviour, the number of execution resources do not have a great impact on the convergence. The number of cycles in the schedule period and the total number of tasks can limit the use of our proposal, but because our solution computes the schedule for a hyper-period (few tens of classical OS ticks, i.e. few tens of milliseconds), this hyper-period allows a large number of neuron evaluations. So, the interest of our solution largely depends on the neural network implementation and on its capability to support a very fast neuron evaluations. In this way, we are currently working on an hardware implementation of our neural network. Due to its very simple computation model, each neuron can be implemented with a very low hardware cost. Remember that we focus on tasks scheduling for sytem-on-chip and that, for this type of systems, a specific hardware block can be developed to ensure a specific functionality. In this particular context, our proposal can be more efficient than other classical solutions.

8. Conclusion

In this paper, a new neural network model is proposed to facilitate the implementation of the scheduling service in the context of heterogeneous multiprocessor architectures. Previous neural network models managing heterogeneous architectures have been adapted from the Hopfield model. They use a very large number of neurons and the major drawback of these solutions is the difficulty to converge toward a valid solution. The major contributions of this paper concern the limitation of hidden neurons and the limitation of the number of ANN re-initializations to ensure the convergence. To limit the number of neurons, we propose to replace the hidden neurons of classical solutions by several inhibitor neurons. With a particular connectivity between these specific neurons and the rest of the neural network, we can limit the number of invalid solutions generated. Based on these inhibitors, a new structure of a neural network is presented. The main characteristic is its capacity to always ensure the rapid convergence towards a valid solution.

We have validated our proposal through simulations, and have shown the efficiency of our proposition compared to previous works. Such a network allows us to reduce significantly the number of neurons necessary to model the problem, since approximatively two times less neurons are used than in the classical case. The other main improvement of our proposal is the substantial reduction (up to 200) in the number of cycles needed to ensure convergence.

These contributions are important advances for our future works on defining an hardware implementation of the scheduling service in the context of SoC architecture. Globally, the limitation of the network size ensures that the hardware implementation will be as simple as possible, and thus as fast as possible. Furthermore, the limitation of the number of network re-initializations is also an interesting point to ensure that the convergence is fast.

From this neural network model, we are currently working on the hardware implementation of such a structure. From the application's characteristics, we produce a register–transfer level VHDL description of the neural network which can be synthetised in an FPGA circuit. Our preliminary results using Xilinx FPGA circuits confirm that neural network convergence is reached in a few clock cycles. Therefore, in the context of SoC architectures, this type of structure has the capacity to compute efficiently the scheduling of several tens of tasks in a few cycles.

References

[1] Wayne Wolf, The future of multiprocessor systems-on-chips, in: DAC '04: Proceedings of the 41st annual Design Automation Conference, ACM, New York, NY, USA, 2004, pp. 681–685.

[2] H. Walder, M. Platzner, Reconfigurable hardware operating systems: from design concepts to realizations, in: Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA), 2003, pp. 284–287.

[3] C. Cardeira, Z. Mammeri, Preemptive and non-preemptive real-time scheduling based on neural networks, in: Proceedings of Distributed Computer Control Systems, Toulouse, France, 1995, pp. 67–72.

[4] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM 20 (1) (1973) 46–61.

[5] J.Y.-T. Leung, J. Whitehead, On the complexity of fixed-priority scheduling of periodic real-time tasks, Performance Evaluation 2 (4) (1982) 237–250.

[6] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.

[7] K. Schild, J. Würtz, Off-line scheduling of a real-time system, in: Proceedings of the ACM Symposium on Applied Computing, Atlanta, USA, 1998, pp. 29–38.

[8] X. Lu, R. Sitters, L. Stougie, A class of on-line scheduling algorithms to minimize total completion time, Operations Research Letters 31 (3) (2003) 232–236.

[9] N. Megow, A.S. Schulz, On-line scheduling to minimize average completion time revisited, Operations Research Letters 32 (5) (2004) 485–490.

[10] M. Young, L.C. Shu, Hybrid online/offline scheduling for hard real-time systems, in: Proceedings of the International Symposium on Real-Time and Media Systems, Taipei, Taiwan, 1996, pp. 231–240.

[11] A. Srinivasan, P. Holman, J.H. Anderson, S. Baruah, The case for Fair multiprocessor scheduling, in: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Washington, DC, USA, 2003, pp. 1143–1153.

[12] S. Baruah, N. Fisher, The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems, IEEE Transactions on Computers 55 (7) (2006) 918–923.

[13] P. Kohout, B. Ganesh, B. Jacob, Hardware support for real-time operating systems, in: Proceedings of the Conference on Hardware/Software Codesign and System Synthesis, Newport Beach, USA, 2003, pp. 45–51.

[14] P. Kuacharoen, M. Shalan, V. Mooney III, A configurable hardware scheduler for real-time systems, in: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, 2003, pp. 96–101.

[15] A. Morton, W.M. Loucks, A hardware/software kernel for system on chip designs, in: Proceedings of the ACM Symposium on Applied Computing, Nicosia, Cyprus, 2004, pp. 869–875.

[16] C. Cardeira, M. Silva, Z. Mammeri, Handling precedence constraints with neural network based real-time scheduling algorithms, in: Proceedings of Euromicro Workshop on Real Time Systems, Toldeo, Spain, 1997, pp. 207–214.

[17] R. Rojas, Neural Networks – A Systematic Introduction, Springer-Verlag, 1996.

[18] G. Tagliarini, J.F. Christ, W.E. Page, Optimization using neural networks, IEEE Transactions on Computers 40 (12) (1991) 1347–1358.

[19] J.J. Hopfield, D.W. Tank, Neural computation of decisions in optimization problems, Biological Cybernetics 52 (3) (1985) 141–152.

[20] M.A. Cohen, S. Grossberg, Absolute stability of global pattern formation and parallel memory storage by competitive neural networks, IEEE Transactions on Systems, Man, and Cybernetics 13 (5) (1983) 815–826.

[21] S. Grossberg, Studies of Mind and Brain: Neural Principles of Learning, Perception, Development, Cognition, and Motor Control, Reidel Press, Boston, 1982.

[22] D. Chillet, I. Benkermi, S. Pillement, O. Sentieys, Hardware task scheduling for heteregeneous SoC architectures, in: Proceedings of the European Signal Processing Conference, Poznan, Poland, 2007, pp. 1653–1657.

[23] D. Chillet, S. Pillement, O. Sentieys, A neural network model for real-time scheduling on heterogeneous SoC architectures, in: Proceedings of the IEEE International Joint Conference on Neural Networks, Orlando, FL, USA, 2007, pp. 102–107.

[24] Dalia Aoun, Anne-Marie Déplanche, Yvon Trinquet, Pfair scheduling improvement to reduce interprocessor migrations, in: Giorgio Buttazzo, Pascale Minet (Eds.), 16th International Conference on Real-Time and Network Systems (RTNS 2008), Rennes France, 2008, pp. 131–138.

[25] Robert P. Dick, David L. Rhodes, Wayne Wolf, Tgff: task graphs for free, in: CODES/CASHE '98: Proceedings of the 6th International Workshop on Hardware/software Codesign, IEEE Computer Society, Washington, DC, USA, 1998, pp. 97–101. <http://ziyang.eecs.umich.edu/dickrp/tgff/>.

Daniel Chillet received the Engineering degree and the M.S. degree in electronics and signal processing engineering from ENSSAT, University of Rennes, respectively, in 1992 and in 1994, and the Ph.D. degree in signal processing and telecommunications from the University of Rennes, in 1997. He is currently an Associate Professor of electrical engineering at the University of Rennes and a member of the CAIRN Research Team at the IRISA Laboratory. His research interests include memory hierarchy, reconfigurable resources, real-time systems, and middleware. All these topics are studied in the context of SoC design for embedded systems.

Antoine Eiche received the Master degree of computer science in 2008 from University Louis Pasteur at Strasbourg. He is currently a Ph.D. student in signal processing and telecommunication at the University of Rennes. He is a member of the CAIRN INRIA team at the IRISA laboratory. His research interests include operating systems with a focus on real-time scheduling, reconfigurable architectures, and artificial neural networks.

Sébastien Pillement is an associate professor in the IUT de Lannion at Rennes 1 University. He is also a research member in the CAIRN INRIA team of the IRISA Lab at Rennes 1 University. He received a Ph.D. degree in Computer Science from the University of Montpellier II. His research interests include Dynamically Reconfigurable architectures, SoC (system-on-chips) design methodology and NoC (Network on Chip) based circuits. He focus his research on designing flexible and efficient architectures managed in real-time.

Olivier Sentieys received the M.Sc. and Ph.D. degrees in Electrical Engineering from the University of Rennes, in 1990 and 1993, respectively. After completing his Habilitation thesis in 1999, he joined University of Rennes (ENSSAT) and IRISA Laboratory, France, as a full Professor of Electronics Engineering, in 2002. He is leading the CAIRN Research Team at INRIA Institute (national institute for research in computer science and control) and is a Cofounder of Aphycare Technologies, a company developing smart sensors for biomedical applications. His research interests include design of mobile communication systems, finite arithmetic effects, low-power and reconfigurable architectures, and cooperation in mobile systems. He is the author or coauthor of more than 80 journal publications or published conference papers and holds six patents.