Parallel Deadlock Detection Algorithm

Abstract

The disadvantages (i), (ii) and (iii) mentioned in Section 3.3.3 unfortunately make the implementation of deadlock avoidance difficult in real systems. Our novel approach to mixing deadlock detection and avoidance (thus, not requiring advanced, a priori knowledge of resource requirements) contributes to easier adaptation of deadlock avoidance in an MPSoC by accommodating maximum freedom (i.e. maximum concurrency of requests and grants depending on a particular execution trace) with the advantage of deadlock avoidance.

The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock. In the case of livelock resulting from attempts to avoid deadlock, the DAU asks one of the processes involved in the livelock to release resource(s) so that the livelock can also be resolved.

Although many deadlock avoidance approaches have been introduced so far [21, 25, 26, 30], to the best of our knowledge, there has been no prior work in a hardware implementation of deadlock avoidance. The DAU not only provides a solution to both deadlock and livelock but is also up to 312× faster than an equivalent software solution (please see the details in Section 5).

In the following few Sections, we further describe these new approaches in more detail.

4.2 New deadlock detection methodology

4.2.1 Parallel deadlock detection algorithm:

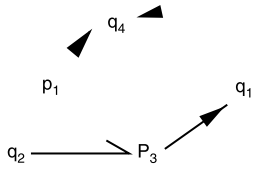
The parallel deadlock detection algorithm (PDDA) dramatically reduces deadlock detection time by mapping a resource allocation graph (RAG [22]; its state is denoted as γ_{ij} [29]) onto a matrix $\boldsymbol{M}_{ij}$ that will have exactly the same request and grant edges as the RAG has but with another notation for each edge. We define a RAG matrix and a terminal reduction sequence before introducing PDDA that exploits the terminal reduction sequence.

Definition 6: The purpose of this definition is to define matrices that correspond to graph γ, system γ_i and state γ_{ij} [29]. A RAG matrix $\boldsymbol{M}$ is a matrix mapped from a RAG γ and represents an arbitrary system with processes and resources. A system matrix $\boldsymbol{M}_i$ is defined as a matrix representation of a particular system γ_i, where the rows (fixed in size) of matrix $\boldsymbol{M}_i$ represent the fixed set Q of resource nodes of γ_i, and the columns (fixed in size) of matrix $\boldsymbol{M}_i$ represent the fixed set P of process nodes of γ_i. We denote another notation of this relationship as $\boldsymbol{M}_i \equiv \gamma_i$ for the sake of simplicity. A state matrix $\boldsymbol{M}_{ij}$ is a matrix that represents a particular system state γ_{ij}, i.e. $\boldsymbol{M}_{ij} \equiv \gamma_{ij}$. Edges E (consisting of request edges R and grant edges G [29]) in system state γ_{ij} are mapped onto the corresponding array elements using the following rule:

Given $E = \{R \cup G\}$ from γ_{ij},

$$\boldsymbol{M}_{ij} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & \alpha_{st} & \vdots \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn} \end{bmatrix}$$

for all rows $1 \leq s \leq m$ and for all columns $1 \leq t \leq n$:



$\boldsymbol{M}_{ij}$ =

$\boldsymbol{M}_{ij}$ =	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6		g		r		

Fig. 11 *Matrix representation example*

$\alpha_{st} = g_{s \to t}$ (or simply 'g'),
if there exists a grant edge $(q_s, p_t) \in G$
$\alpha_{st} = r_{t \to s}$ (or simply 'r'),
if there exists a request edge $(p_t, q_s) \in R$
$\alpha_{st} = 0_{st}$ ('0' or a blank space), otherwise,

where m and n are the numbers of resources and processes, respectively.

Example 3: State matrix representation

The system state γ_{ij} shown on the upper half of Fig. 11 can be represented in the matrix form shown in the bottom half of Fig. 11.

Based on a state matrix $\boldsymbol{M}_{ij}$, instead of finding an exact cycle (as other algorithms do, e.g. see Chap. 4 of [22]), PDDA removes edges that have nothing to do with cycles; this edge removal process is called a terminal reduction sequence. After the terminal reduction sequence (e.g. using k edge removal steps) removes all reducible edges (resulting in an 'irreducible' matrix $\boldsymbol{M}_{i,j+k}$), if edges still exist, then deadlock(s) exist. On the other hand, if $\boldsymbol{M}_{ij}$ has been completely reduced, no deadlock exists. Intuitively, removing reducible edges corresponds to the best sequence of operations a particular process can execute to help unblock other processes. Before describing the terminal reduction sequence in detail, we define what we mean by 'terminal' in different uses.

Definition 7: A terminal row τ_s is a row s (recall that row s corresponds to resource q_s) of matrix $\boldsymbol{M}_{ij}$ such that either (i) all non-zero entries $\{\alpha_{st_r} \neq 0, 1 \leq t_r \leq n\}$ are request entries $r_{t_r \to s}$ with at least one request entry (i.e. one or more request entries and no grant entry in the row), or (ii) one entry $\alpha_{st_g}, 1 \leq t_g \leq n$, is a grant $g_{s \to t_g}$ with the rest of the entries $\{\alpha_{st}, 1 \leq t \leq n, t \neq t_g\}$ equal to zero.

Definition 8: A terminal column τ_t is a column t (recall that column t corresponds to process p_t) of matrix $\boldsymbol{M}_{ij}$ such that either (i) all non-zero entries $\{\alpha_{st} \neq 0, 1 \leq s \leq m\}$ are request entries with at least one request entry (i.e. one or more request entries and no grant entry in the column),

or (ii) all nonzero entries $\{\alpha_{st} \neq 0, 1 \leq s \leq m\}$ are grant entries with at least one grant entry (i.e. one or more grant entries and no request entry in the column).

Definition 9: An edge that belongs to either a terminal row τ_s or a terminal column τ_t is called a terminal edge.

The next definition defines one step of a terminal reduction sequence.

Definition 10: A terminal reduction step ϵ is a unary operator $\epsilon : \boldsymbol{M}_{ij} \mapsto \boldsymbol{M}_{i,j+1}$, where ϵ calculates the terminal edge set and returns $\boldsymbol{M}_{i,j+1}$ such that all terminal edges found are removed by setting the terminal entries found to zero; thus, the next iteration $\boldsymbol{M}_{i,j+1}$ will start with equal or fewer total edges as compared to $\boldsymbol{M}_{ij}$. This terminal reduction step is denoted as $\epsilon(\boldsymbol{M}_{ij})$, i.e. $\boldsymbol{M}_{i,j+1} = \epsilon(\boldsymbol{M}_{ij})$.

Note that the removals of terminal edges in $\boldsymbol{M}_{ij}$ enable the discovery of new terminal nodes in $\boldsymbol{M}_{i,j+1}$. Any new terminal nodes that appear were connect nodes in $\boldsymbol{M}_{ij}$ that were connected to terminal nodes in $\boldsymbol{M}_{ij}$.

Example 4: One step of terminal reduction (ϵ)
Figure 12*b* shows a new matrix $\boldsymbol{M}_{i,j+1}$ after a matrix reduction step ϵ, defined in Definition 10, is applied to M_{ij} shown in (a). In matrix $\boldsymbol{M}_{ij}$, since q_2 and q_3 are terminal rows by Definition 7, all the edges in their rows are terminal edges. Therefore, all the edges in rows q_2 and q_3 can be removed. Likewise, p_2, p_4 and p_6 are terminal columns by Definition 8; hence, all edges in these columns can be removed, resulting in matrix $\boldsymbol{M}_{i,j+1}$.

Definition 11: A terminal reduction sequence ξ, applicable to a matrix $\boldsymbol{M}_{ij}$, is a sequence of k terminal reduction steps ϵ (recall that ϵ is a terminal reduction step) such that: (i) $\boldsymbol{M}_{ij} \mapsto \boldsymbol{M}_{i,j+1} \mapsto \cdots \mapsto \boldsymbol{M}_{i,j+k}$; (ii) $\boldsymbol{M}_{i,j+k}$ is irreducible (i.e. $\epsilon(\boldsymbol{M}_{i,j+k}) = \boldsymbol{M}_{i,j+k}$); and (iii) $\{\boldsymbol{M}_{i,j+h}, 0 \leq h < k\}$ are all unique and reducible. A terminal reduction sequence is called a complete reduction when the sequence of terminal reduction steps corresponding to ξ results in $\boldsymbol{M}_{i,j+k}$ such that the irreducible state matrix $\boldsymbol{M}_{i,j+k}$ contains all zero entries (note that this means that $\gamma_{i,j+k}$ corresponding to $\boldsymbol{M}_{i,j+k}$ has no edges: $E(\gamma_{i,j+k}) = \emptyset$). A terminal reduction sequence is called an incomplete reduction when ξ returns $\boldsymbol{M}_{i,j+k}$ with at least one non-zero entry (note that this means that $\gamma_{i,j+k}$ corresponding to $\boldsymbol{M}_{i,j+k}$ has at least one edge: $E(\gamma_{i,j+k}) \neq \emptyset$). Another representation of a terminal reduction sequence is shown in (1):

$$\begin{aligned} \boldsymbol{M}_{i,j+k} &= \xi(\boldsymbol{M}_{ij}) \\ &= \epsilon^{(k)}(\ldots \epsilon^{(2)}(\epsilon^{(1)}(\boldsymbol{M}_{ij}))\ldots) \\ &= \epsilon(\ldots \epsilon(\epsilon(\boldsymbol{M}_{ij}))\ldots) \end{aligned} \quad (1)$$

We now introduce two algorithms, one being a terminal reduction sequence algorithm that implements the terminal reduction sequence ξ, the other being PDDA, which employs the terminal reduction sequence algorithm.

Algorithm 1 is an implementation of the terminal reduction sequence ξ shown in Definition 11. We summarise the operation of Algorithm 1. Lines 2 and 3 of Algorithm 1 initialise two variables: iterator k and matrix $\boldsymbol{M}_{iter}$, which is a copy of an input argument $\boldsymbol{M}_{ij}$. Line 5 finds all terminal rows (Definition 7), and line 6 finds all terminal columns (Definition 8). Line 7 checks whether M_{iter} has more terminal edges, and, if no more terminal edges exist, the current iteration ends. Lines 8 and 9 remove all the terminal edges found at the current iteration. On the whole, the terminal reduction step $\epsilon(\boldsymbol{M}_{ij})$ of Definition 10 corresponds to lines 5–9 of Algorithm 1, which iterates until the matrix $\boldsymbol{M}_{iter}$ becomes irreducible. Note that, in hardware implementation, lines 5 and 6 of Algorithm 1 are executed at the same time, as are lines 8 and 9.

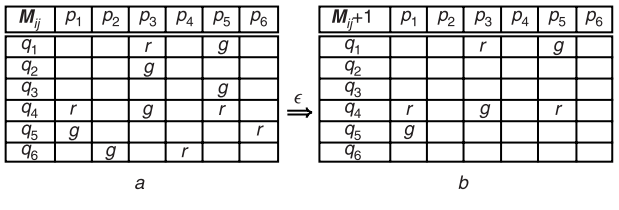


Fig. 12 *One terminal reduction step* (ϵ) *example*

Algorithm 1: terminal reduction sequence algorithm

```
1 ξ(M_ij){
2    k = 0;
3    M_iter = M_ij;
4    while (1) {
       /* parallel on */
5      calculate τ_s for all s; /* terminal rows */
6      calculate τ_t for all t; /* terminal columns */
       /* parallel off */
7      if (neither ∃τ_s nor ∃τ_t) break;
         /* if no more terminals */
       /* parallel on */
8      for all s such that ∃τ_s,
         set all entries in row s of M_iter to zero;
9      for all t such that ∃τ_t,
         set all entries in column t of M_iter to zero;
       /* parallel off */
10     k = k + 1;
11   }
12   M_{i,j+k} = M_iter;
13   return M_{i,j+k};
14 }
```

Algorithm 2: Parallel deadlock detection algorithm

```
1 PDDA (γ_ij) {
2      M[s,t] = [α_st], where
3        s = 1,...,m and t = 1,...,n
4        α_st = r, if ∃(p_t, q_s) ∈ E(γ_ij)
5        α_st = g, if ∃(q_s, p_t) ∈ E(γ_ij)
6        α_st = 0, otherwise.
7      M_{i,j+k} = ξ(M_ij); /* call Algorithm 1 */
8      if (M_{i,j+k} == [0]) {/* if matrix of all zeros */
9        return 0; /* no deadlock */
10     } else {
11       return 1; /* deadlock detected */
12     }
13 }
```

We now summarise the operation of Algorithm 2. Lines 2–6, given γ_{ij}, construct the corresponding matrix $\boldsymbol{M}_{ij}$ according to Definition 6. Next, line 7 calls Algorithm 1 with argument $\boldsymbol{M}_{ij}$. When Algorithm 1 is completed, lines 8–12 of Algorithm 2 determine whether γ_{ij} has a deadlock or not by considering returned matrix $\boldsymbol{M}_{i,j+k}$: if $\boldsymbol{M}_{i,j+k}$ is empty, the corresponding γ_{ij} has no deadlock; otherwise, deadlock(s) exist. Finally, Algorithm 2 returns '1' if the system state under consideration has deadlock(s); otherwise, Algorithm 2 returns '0' indicating no deadlock exists. Note that Algorithm 2, which includes Algorithm 1, is referred to as PDDA.

We have proven that PDDA detects deadlock if and only if there exists a cycle in state γ_{ij} [29]. We have also proven

that our hardware implementation of Algorithm 1 completes its computation in at most $2 \times \min(m,n) - 3 = O(\min(m,n))$ steps, where m is the number of resources and n is the number of processes [29].

4.2.2 Hardware implementation of PDDA: DDU:

We here summarise the operation of PDDA in the hardware point of view, i.e. how to parallelise PDDA to implement in hardware (please see [29] for more information, which describes the sequence of DDU operations in great detail). As introduced in Section 4.2.1., a given system state γ_{ij} is equivalently represented by a system state matrix $\boldsymbol{M}_{ij}$ (shown in equation 2) so that, based on $\boldsymbol{M}_{ij}$, the DDU can perform the sequence of operations shown in Algorithm 1 and 2 and decide whether the given state has a deadlock or not:

$$\boldsymbol{M}_{ij} = \begin{bmatrix} \alpha_{11} & \cdots & \alpha_{1t} & \cdots & \alpha_{1n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha_{s1} & \cdots & \alpha_{st} & \cdots & \alpha_{sn} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha_{m1} & \cdots & \alpha_{mt} & \cdots & \alpha_{mn} \end{bmatrix} = \boldsymbol{M}_{iter} \tag{2}$$

where m is the number of resources and n is the number of processes.

Each matrix element α_{st} in $\boldsymbol{M}_{ij}$ represents one of the following: $g_{s\to t}$ (a grant edge), $r_{t\to s}$ (a request edge) or 0_{st} (no edge). Since α_{st} is ternary-valued, α_{st} can be minimally defined as a pair of two bits $\alpha_{st} = (\alpha^r_{st}, \alpha^g_{st})$. If an entry α_{st} is a grant edge g, bit α^r_{st} is set to 0, and α^g_{st} is set to 1; if an entry α_{st} is a request edge r, bit α^r_{st} is set to 1, and α^g_{st} is set to 0; otherwise, both bits α^r_{st} and α^g_{st} are set to 0. Hence, an entry α_{st} can be only one of the following binary encodings: 01 (a grant edge), 10 (a request edge) or 00 (no activity). Thus, $\boldsymbol{M}_{iter}$ in line 3 of Algorithm 1 can be written as shown in (3):

$$\boldsymbol{M}_{iter} = \begin{bmatrix} (\alpha^r_{11}, \alpha^g_{11}) & \cdots & (\alpha^r_{1t}, \alpha^g_{1t}) & \cdots & (\alpha^r_{1n}, \alpha^g_{1n}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^r_{s1}, \alpha^g_{s1}) & \cdots & (\alpha^r_{st}, \alpha^g_{st}) & \cdots & (\alpha^r_{sn}, \alpha^g_{sn}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^r_{m1}, \alpha^g_{m1}) & \cdots & (\alpha^r_{mt}, \alpha^g_{mt}) & \cdots & (\alpha^r_{mn}, \alpha^g_{mn}) \end{bmatrix} \tag{3}$$

Finding terminal rows and terminal columns, which corresponds to lines 5 and 6 of Algorithm 1, requires three logical operations performed in sequence: (i) bit-wise-or (BWO); (ii) eXclusive-OR (XOR); and (iii) OR. Two separate BWO operations, shown in (4), take place through each row and each column of $\boldsymbol{M}_{iter}$, all in parallel at the same time at each iteration in the DDU:

$$\begin{aligned} BWO^c_{iter} &= \forall t, (\alpha^r_{ct}, \alpha^g_{ct}) = \forall t, \left(\bigvee_{s=1}^{m} \alpha^r_{st}, \bigvee_{s=1}^{m} \alpha^g_{st} \right) \\ BWO^r_{iter} &= \forall s, (\alpha^r_{rs}, \alpha^g_{rs}) = \forall s, \left(\bigvee_{t=1}^{m} \alpha^r_{st}, \bigvee_{t=1}^{m} \alpha^g_{st} \right) \end{aligned} \tag{4}$$

where notation $\forall$ means for all and notation $\bigvee$ means bit-wise-or of elements.

Then, from the results of two BWO operations, the XOR operations, shown in (5), for each row and each column occur all in parallel:

$$\begin{aligned} XOR^c_{iter} &= \forall t, \tau_{ct} = \forall t, (\alpha^r_{ct} \oplus \alpha^g_{ct}) \\ XOR^r_{iter} &= \forall s, \tau_{rs} = \forall s, (\alpha^r_{rs} \oplus \alpha^g_{rs}) \end{aligned} \tag{5}$$

where $\oplus$ denotes eXclusive-OR.

Next, the OR operation, shown in (6), produces a termination condition (i.e. the reducibility test of matrix $\boldsymbol{M}_{iter}$, which corresponds to line 7 in Algorithm 1) at each iteration. That is, the termination condition represents whether a current matrix is further reducible or not. If T_{iter} equals '1', meaning that more terminal edge(s) exist, the iterations continue. If the current matrix $\boldsymbol{M}_{iter}$ is irreducible (i.e. it has no terminal edges), T_{iter} will become '0'; thus, further iterations would accomplish nothing. This irreducibility condition can be written as

$$T_{iter} = (\tau_C \vee \tau_R) = \left(\bigvee_{t=1}^{n} \tau_{ct} \vee \bigvee_{s=1}^{m} \tau_{rs} \right) \tag{6}$$

Before finishing PDDA, one more important process remains: deadlock detection, which requires two more parallel logic operations. Equation (7) represents the existence of connect nodes in each column and in each row, respectively, involved in cycle(s):

$$\begin{aligned} AND^c_{iter} &= \forall t, \phi_{ct} = \forall t, (\alpha^r_{ct} \wedge \alpha^g_{ct}) \\ AND^r_{iter} &= \forall s, \phi_{rs} = \forall s, (\alpha^r_{rs} \wedge \alpha^g_{rs}) \end{aligned} \tag{7}$$

where $\wedge$ denotes bit-wise-and of elements.

Finally, (8) produces the result of deadlock detection, which corresponds to lines 8–12 of Algorithm 2:

$$D_{iter} = (\phi_c \vee \phi_r) = \left(\bigvee_{t=1}^{n} \phi_{ct} \vee \bigvee_{s=1}^{m} \phi_{rs} \right) \text{ when } T_{iter} = 0 \tag{8}$$

4.2.3 Architecture of deadlock detection unit:

The DDU consists of three parts as shown in Fig. 13: matrix cells, weight cells and a decide cell. Part 1 is the system state matrix $\boldsymbol{M}_{ij}$ consisting of an array of matrix cells α_{st}. Part 2 consists of two weight vectors: (i) one column weight vector below the matrix cells and (ii) one row weight vector on the right side of matrix cells. The column weight vector is expressed as follows:

$$W^c = [\, w_{c1} \quad w_{c2} \quad \cdots \quad w_{ct} \quad \cdots \quad w_{cn} \,] \tag{9}$$

where n is the number of processes, and $\forall t, w_{ct}$ (each column weight cell) is a pair (τ_{ct}, ϕ_{ct}), representing whether

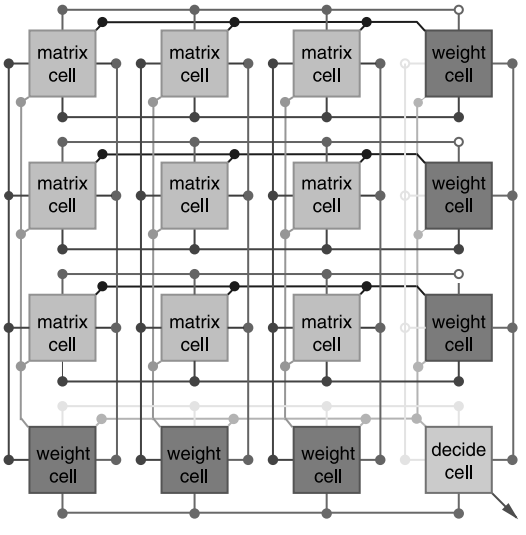


Fig. 13 *DDU architecture*

Table 1: Synthesis results for DDU

No. of processes × no. of resources	Lines of Verilog	Area in terms of two-input NAND gates	Worst case no. of iterations
2 × 3	49	186	2
5 × 5	73	364	6
7 × 7	102	455	10
10 × 10	162	622	16
50 × 50	2682	14142	96

the corresponding process node is a terminal node (1, 0), a connect node (0, 1), or neither (0, 0). The row weight vector is expressed as follows:

$$\boldsymbol{W}^r = [\, w_{r1} \quad w_{r2} \quad \cdots \quad w_{rs} \quad \cdots \quad w_{rm} \,]^{\mathrm{T}} \qquad (10)$$

where m is the number of resources, and $\forall s, w_{rs}$ (each row weight cell) is a pair (τ_{rs}, ϕ_{rs}), representing whether the corresponding resource node is a terminal node, a connect node or neither. Part 3 is one decide cell D_{iter} at the bottom right corner of the DDU.

Figure 13 shows the architecture of the DDU for three processes and three resources. This DDU example has nine matrix cells (3×3) for all edge elements of $\boldsymbol{M}_{ij}$, six weight cells (three for column processing and three for row processing), and one decide cell for making the decision of deadlock.

4.2.4 Synthesis results for DDU:

We used the Synopsys Design Compiler (DC) to synthesise the DDU with a 0.3 µm standard cell library from AMIS [31]. Table 1 shows the synthesis results of five types of DDUs customised according to the number of processes and resources in an SoC. The fourth column, denoted 'worst case no. of iterations', represents the number of worst case number of iterations for the corresponding DDU.

Please note that a system example using the DDU, including quantitative performance results, will be presented in Section 5.3.

4.3 New deadlock avoidance methodology

In our new approach to deadlock avoidance, we utilise the parallel deadlock detection algorithm (PDDA) and DDU. Unlike the DDU, we have thought that it would be very helpful if there were a hardware unit that not only detects deadlock but also avoids possible deadlock within a few clock cycles and with a small amount of hardware.

The deadlock avoidance unit (DAU), if employed, tracks all requests and releases of resources. In other words, the DAU receives, interprets and executes commands from processes; then it returns DAU processing results back to processes. The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock.

4.3.1 New deadlock avoidance algorithm:

Algorithm 3 shows our deadlock avoidance approach. We initially considered two other deadlock avoidance approaches but found Algorithm 3 to be better because it resolves livelock more actively and efficiently than two other approaches [28].

Let us proceed to describe Algorithm 3 step by step. When a process requests a resource from the DAU (line 2 of Algorithm 3), the DAU checks for the availability of the resource requested (line 3). If the resource is available (i.e. no one is using it), the resource will be granted to the requester immediately (line 4). If the resource is not available, the DAU check the possibility of request deadlock (R-dl) (line 5). If a request would cause request deadlock (R-dl) (line 5) – note that the DAU tracks all requests and releases – the DAU compares the priority of the requester with that of the current owner of the requested resource. If the priority of the requester is higher than that of the current owner of the resource (line 6), the DAU makes the request be pending for the requester (line 7), an then the DAU asks the owner of the resource to give up the resource so that the higher priority process can proceed (line 8, the current owner may need time to finish or checkpoint its current processing). On the other hand, if the priority of the requester is lower than that of the owner of the resource (line 9), the DAU asks the requester to give up the resource(s) that the requester already has but is most likely not using yet (since all needed resources are not yet granted, line 10).

Algorithm 3: Deadlock avoidance algorithm (DAA)

```
DAA (event) {
 1   case (event) {
 2     a request:
 3       if the resource is available
 4         grant the resource to the requester
 5       else if the request would cause request deadlock
           (R-dl)
 6         if the priority of the requester greater than that
             of the owner
 7           make the request be pending
 8           ask the current owner of the resource to
               release the resource
 9         else
10           ask the requester to give up resource(s)
11         end-if
12       else
13         make the request be pending
14       end-if
15       break
16     a release:
17       if any process is waiting for the released resource
18         if the grant of the resource would cause grant
             deadlock
19           grant the resource to a lower priority process
               waiting
20         else
21           grant the resource to the highest priority
               process waiting
22         end-if
23       else
24         make the resource become available
25       end-if
26   } end-case
}
```

When the DAU receives a resource release command from a process (line 16) and any process is waiting for the resource (line 17), before actually granting the released resource to one of the requesters, the DAU temporarily marks a grant of the resource to the highest priority process (on its internal matrix). Then, to check potential grant deadlock, the DAU executes its deadlock detection algorithm. If the temporary grant does not cause grant deadlock (G-dl) (line 20), it becomes a fixed grant; thus the resource is granted to the highest priority requester (line 21). On the other hand, if the temporary grant causes G-dl (line 18), the temporary grant will be undone; then, because the released resource cannot

References

Alpaydn, G., & Dundar, G. (2002). Evolution-based design of neural fuzzy networks using self-adaptive genetic parameters. *IEEE Transactions on Fuzzy Systems*, *10*, 211–221.

Boutails, Y., Kottas, L. T., & Christodoulou, M. (2009). Adaptive estimation of fuzzy cognitive maps with proven stability and parameter convergence. *IEEE Transactions on Fuzzy Systems*, *17*, 874–889.

Chun, F. H. (2007). Self-organizing adaptive fuzzy neural control for a class of nonlinear systems. *IEEE Transactions on Fuzzy Systems*, *18*, 1232–1241.

Dudul, S. V. (2005). Prediction of Lorenz chaotic attractor using two-layer perceptron neural network. *Applied Soft Computing*, *5*, 333–355.

Gaweda, A. E., & Zurada, J. M. (2003). Data-driven linguistic modeling using relational fuzzy rules. *IEEE Transactions on Fuzzy Systems*, *11*, 121–134.

Hou, S. M., & Li, Y. R. (2009). Short-term fault prediction based on support vector machines with parameter optimization by evolution strategy. *Expert Systems with Applications*, *36*, 12383–12391.

Juang, C., & Tsao, Y. (2008). A self-evolving interval type-2 fuzzy neural network with on-line structure and parameter learning. *IEEE Transactions on Fuzzy Systems*, *16*, 1411–1424.

Jun, Z., Henry, S. H. C., & Lo, W. L. (2008). Chaotic time series prediction using a neuro-fuzzy system with time-delay coordinates. *IEEE Transactions on Knowledge and Data Engineering*, *20*, 956–964.

Kasabov, N. K., & Song, Q. (2002). DENFIS: dynamic evolving neural-fuzzy inference system and its application for time-series prediction. *IEEE Transactions on Fuzzy Systems*, *10*, 144–154.

Kosko, B. (1986). Fuzzy cognitive maps. *International Journal of Man–Machine Studies*, *24*, 65–75.

Kosko, B. (1997). *Fuzzy engineering*. Englewood Cliffs, NJ: Prentice-Hall.

Kottas, F. L., & Boutalis, Y. S. (2004). A new method for reaching equilibrium points in fuzzy cognitive maps. In *Proc. 2nd international IEEE conference on intelligent sys. Vol. 1* (pp. 53–60).

Kottas, T. L., & Boutalis, Y. S. (2006). New maximum power point tracker for PV arrays using fuzzy controller in close cooperation with fuzzy cognitive networks. *IEEE Transactions on Energy Conversion*, *21*, 793–803.

Shi, Z. W., & Han, M. (2007). Support vector echo-state machine for chaotic time-series prediction. *IEEE Transactions on Neural Networks*, *18*, 359–371.

Song, H. J., Miao, C. Y., & Shen, Z. Q. (2009). A fuzzy neural network with fuzzy impact grades. *Neurocomputing*, *72*, 3098–3122.

Song, H. J., Shen, Z. Q., & Miao, C. Y. M. (2007). Fuzzy cognitive map learning based on multi-objective particle swarm optimization. In *The 9th annual conference on genetic and evolutionary computation* (p. 69).

Stach, W. S., Kurgan, L. A., & Pedrycz, W. (2008). Numerical and linguistic prediction of time series with the use of fuzzy cognitive maps. *IEEE Transactions on Fuzzy Systems*, *16*, 61–72.

Stylios, C. D., & Groumpos, P. P. (1999). Fuzzy cognitive maps: a model for intelligent supervisory control systems. *Computers in Industry*, *39*, 229–238.

Stylios, C. D., & Groumpos, P. P. (2004). Modeling complex systems using fuzzy cognitive maps. *IEEE Transactions on Systems, Man, and Cybernetics—Part A*, *34*, 155–162.

Zhang, J. Y., Liu, Z. Q., & Zhou, S. (2003). Quotient FCMs—a decomposition theory for fuzzy cognitive maps. *IEEE Transactions on Fuzzy Systems*, *11*, 593–604.

Zhang, J. Y., Liu, Z. Q., & Zhou, S. (2006). Dynamic domination in fuzzy causal networks. *IEEE Transactions on Fuzzy Systems*, *14*, 42–56.

Zhou, S., Liu, Z. Q., & Zhang, J. Y. (2006). Fuzzy causal networks: general model, inference, and convergence. *IEEE Transactions on Fuzzy Systems*, *14*(Jun), 412–420.

H.J. Song received the Ph.D. degree from Nanyang Technological University (NTU), Singapore, in 2010. He is currently a research fellow in the Emerging Research Lab (ER Lab), Computer Engineering, NTU. From 2008 to 2010, he was researcher at IMEC, Belgium. His research interests include neural networks, fuzzy systems, fuzzy rule-based systems, fuzzy cognitive maps, architecture design methods and system-level exploration for power and memory footprint within real-time constraints.