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 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 $\gamma_y$ [29]) onto a matrix $M_y$ 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 $\gamma$, system $\gamma$, and state $\gamma_y$ [29]. A RAG matrix $M$ is a matrix mapped from a RAG $\gamma$ and represents an arbitrary system with processes and resources. A system matrix $M_i$ is defined as a matrix representation of a particular system $\gamma_i$, where the rows (fixed in size) of matrix $M_i$ represent the fixed set $Q$ of resource nodes of $\gamma_i$, and the columns (fixed in size) of matrix $M_i$ represent the fixed set $P$ of process nodes of $\gamma_i$. We denote another notation of this relationship as $M_i \equiv \gamma_i$ for the sake of simplicity. A state matrix $M_{\gamma_y}$ is a matrix that represents a particular system state $\gamma_y$, i.e. $M_{\gamma_y} \equiv \gamma_y$. Edges $E$ (consisting of request edges $R$ and grant edges $G$ [29]) in system state $\gamma_y$ are mapped onto the corresponding array elements using the following rule:

Given $E = \{R \cup G\}$ from $\gamma_y$,

$$M_{\gamma_y} = \begin{bmatrix}
\gamma_{11} & \gamma_{12} & \cdots & \gamma_{1n} \\
\gamma_{21} & \gamma_{22} & \cdots & \gamma_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
\gamma_{m1} & \gamma_{m2} & \cdots & \gamma_{mn}
\end{bmatrix}$$

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

- $\gamma_{ij} = 0$ if there exists a grant edge $(q_i, q_j) \in G$,
- $\gamma_{ij} = 1$ if there exists a request edge $(p_i, q_j) \in R$,
- $\gamma_{ij} = 0$ (‘0’ or a blank space), otherwise,

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

Example 3: State matrix representation

The state system $\gamma_y$ 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 $M_{\gamma_y}$, 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 $M_{\gamma_y}$), if edges still exist, then deadlock(s) exist. On the other hand, if $M_{\gamma_y}$ 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 $\tau_s$ is a row $s$ (recall that row $s$ corresponds to resource $q_s$) of matrix $M_{\gamma_y}$ such that either (i) all non-zero entries $\{\gamma_{st} \neq 0, 1 \leq t \leq n\}$ are request entries $r_{t \rightarrow s}$ with at least one request entry (i.e. one or more request entries and no grant entry in the row), or (ii) one entry $\gamma_{st}$, $1 \leq t \leq n$, is a grant $g_{s \rightarrow t}$ with the rest of the entries $\{\gamma_{st} \neq 0, 1 \leq t \leq n, t \neq t_s\}$ equal to zero.

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

![Fig. 11 Matrix representation example](image-url)
Definition 9: An edge that belongs to either a terminal row or a terminal column is called a terminal edge.

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

Definition 10: A terminal reduction step  is a unary operator  such that all terminal edges found are removed by setting the terminal entries found to zero; thus, the next iteration  will start with equal or fewer total edges as compared to . This terminal reduction step is denoted as  (Definition 10) is an implementation of the terminal reduction sequence algorithm. Algorithm 1 is an implementation of the terminal reduction sequence algorithm.

Algorithm 1: terminal reduction sequence algorithm

Algorithm 2: Parallel deadlock detection algorithm

We now summarise the operation of Algorithm 2. Lines 2–6, given , construct the corresponding matrix , according to Definition 6. Next, line 7 calls Algorithm 1 with argument . When Algorithm 1 is completed, lines 8–12 of Algorithm 2 determine whether has a deadlock or not by considering returned matrix . If is empty, the corresponding 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 . We have also proven
that our hardware implementation of Algorithm 1 completes its computation in at most $2 \times \min(m, n) = 3 = \Theta(\min(m, n))$ steps, where $m$ is the number of resources and $n$ is the number of processes [29].

4.2.2 Hardware implementation of PDFA: DDU: We here summarise the operation of PDFA in the hardware point of view, i.e., how to parallelise PDFA 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 $\gamma_{ij}$ is equivalently represented by a system state matrix $M_{ij}$ (shown in equation 2) so that, based on $M_{ij}$, 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:

$$M_{ij} = \begin{bmatrix} a_{11} & \cdots & a_{1t} & \cdots & a_{1m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{it} & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{im} & \cdots & a_{mt} & \cdots & a_{mm} \end{bmatrix} = M_{\text{iter}}$$ (2)

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

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

$$M_{\text{iter}} = \begin{bmatrix} (x_{11}^{g}, x_{11}^{s}) & \cdots & (x_{t1}^{g}, x_{t1}^{s}) & \cdots & (x_{1m}^{g}, x_{1m}^{s}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (x_{it}^{g}, x_{it}^{s}) & \cdots & (x_{rt}^{g}, x_{rt}^{s}) & \cdots & (x_{im}^{g}, x_{im}^{s}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (x_{im}^{g}, x_{im}^{s}) & \cdots & (x_{mt}^{g}, x_{mt}^{s}) & \cdots & (x_{mm}^{g}, x_{mm}^{s}) \end{bmatrix}$$ (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 $M_{\text{iter}}$, all in parallel at the same time at each iteration in the DDU:

$$BWO_{\text{iter}}^{r} = \forall t, (x_{it}^{g}, x_{it}^{s}) = \forall t, \left( \bigvee_{i=1}^{m} x_{it}^{g} \bigvee_{t=1}^{m} x_{it}^{s} \right)$$

$$BWO_{\text{iter}}^{c} = \forall s, (x_{rs}^{g}, x_{rs}^{s}) = \forall s, \left( \bigvee_{r=1}^{m} x_{rs}^{g} \bigvee_{t=1}^{m} x_{rs}^{s} \right)$$ (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:

$$\text{XOR}_{\text{iter}}^{r} = \forall t, (x_{it}^{g}, x_{it}^{s}) = \forall t, \left( x_{it}^{g} \bigoplus x_{it}^{s} \right)$$

$$\text{XOR}_{\text{iter}}^{c} = \forall s, (x_{rs}^{g}, x_{rs}^{s}) = \forall s, \left( x_{rs}^{g} \bigoplus x_{rs}^{s} \right)$$ (5)

where $\oplus$ denotes eXclusive-OR.

Next, the OR operation, shown in (6), produces a termination condition (i.e., the reducibility test of matrix $M_{\text{iter}}$, 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 $T_{\text{iter}}$ equals ‘1’, meaning that more terminal edge(s) exist, the iterations continue. If the current matrix $M_{\text{iter}}$ is irreducible (i.e., it has no terminal edges), $T_{\text{iter}}$ will become ‘0’; thus, further iterations would accomplish nothing. This irreducibility condition can be written as

$$T_{\text{iter}} = (\tau_{c} \lor \tau_{g}) = \left( \bigvee_{t=1}^{m} \tau_{ct} \bigvee_{s=1}^{m} \tau_{cs} \right)$$ (6)

Before finishing PDFA, 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):

$$\text{AND}_{\text{iter}}^{r} = \forall t, \phi_{ct} = \forall t, \left( x_{ct}^{g} \land x_{ct}^{s} \right)$$

$$\text{AND}_{\text{iter}}^{c} = \forall s, \phi_{cs} = \forall s, \left( x_{cs}^{g} \land x_{cs}^{s} \right)$$ (7)

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

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

$$D_{\text{iter}} = (\phi_{c} \lor \phi_{t}) = \left( \bigvee_{t=1}^{n} \phi_{ct} \bigvee_{s=1}^{n} \phi_{cs} \right) \text{ when } T_{\text{iter}} = 0$$ (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 $M_{ij}$ consisting of an array of matrix cells $a_{it}$. 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:

$$W_{c} = [w_{c1}, w_{c2}, \ldots, w_{ct}, \ldots, w_{cn}]$$ (9)

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

![Fig. 13  DDU architecture](image-url)
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:

$$
W^r = [w_{r1} \, w_{r2} \ldots \, w_{rs} \, \ldots \, w_{r\text{row}}]^T
$$

where $m$ is the number of resources, and $\forall s$, $w_{rs}$ (each row weight cell) is a pair $(\tau_{rs}, \theta_{rs})$, representing whether the corresponding resource node is a terminal node, a connect node or neither. Part 3 is one decide cell $D_{\text{iter}}$ 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 ($3 \times 3$) for all edge elements of $M_p$, 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 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), and 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)

```plaintext
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


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.