A Novel $O(1)$ Parallel Deadlock Detection Algorithm and Architecture for Multi-unit Resource Systems

Xiang Xiao and Jaehwan John Lee
ECE Department, Purdue School of Engineering and Technology
Indiana University-Purdue University Indianapolis, USA
{xxiao, johnlee}@iupui.edu

Abstract

This paper introduces a novel $O(1)$ parallel deadlock detection approach for multi-unit resource System-on-a-Chips (SoCs), inspired by Kim’s method in $O(1)$ detection as well as Shiu’s method in parallel processing. Our contributions are (i) the first $O(1)$ hardware deadlock detection and (ii) $O(\min(m,n))$ preparation, both for multi-unit resource systems, where $m$ and $n$ are the number of processes and resources, respectively. $O(\min(m,n))$, previously $O(m \times n)$, is achieved by performing all the searches for sink nodes for each and every resource in parallel in hardware over a matrix representing resource allocations as well as other auxiliary matrices. Our experiments demonstrate that deadlock detection always takes two clock cycles.

1 Introduction

Semiconductor technology till now has led to the doubling of transistor counts on a processor chip every 18 months [1]. This technology advance realizes the concept of System-on-a-Chip (SoC) that can integrate several heterogeneous processors and dozens of on-chip hardware resources on a single chip. To exploit parallelism in the multiprocessor hardware, such SoCs allow multiple jobs (processes) to run concurrently on different processors. SoCs often provide multiple units of the same type of resource so that more processors can be satisfied with their resource requests even if the contention for that type of resource is relatively high. While this particular approach can improve the throughput of such SoCs, it not only increases the probability of deadlocks but also makes harder to detect deadlocks. These factors seemingly demand that future SoCs have fast and deterministic deadlock detection service under a multi-unit resource environment.

In this paper, we present a very fast parallel Multi-unit resource Deadlock Detection Algorithm (MDDA), inspired by Kim’s work [4] as well as Shiu’s work [9] and implemented in hardware using Verilog. To enable parallel computation in hardware and cope with multi-unit resource systems, we extend the matrix representation for a Resource Allocation Graph (RAG) from Shiu’s method [9] with several additional matrices. The contributions of this paper are (i) an improved and deterministic $O(1)$ multi-unit resource deadlock detection and (ii) a reduced $O(\min(m,n))$ overall run-time complexity when implemented in hardware, where $m$ is the number of processes and $n$ is the number of resources. Kim’s multi-unit resource deadlock detection algorithm has an overall run-time complexity of $O(m \times n)$ [4]. Relationships between our algorithm and others are stated in detail in the following section.

2 Related Work

A set of processes is deadlocked if every process in the set is indefinitely waiting for certain resources that only other processes in the same set can release. There have been a variety of deadlock detection algorithms proposed in the past. A deadlock detection algorithm usually assumes that its target system contains either only single-unit resources (single-unit resource systems) or single-unit as well as multi-unit resources (multi-unit resource systems).

For single-unit resource systems, Kim and Koh [5] described an algorithm with an $O(1)$ deadlock detection run-time and an $O(m + n)$ overall run-time complexity, where $m$ and $n$ are equal to the number of processes and resources in the system, respectively. Shiu et al. [9] presented a parallel algorithm for single-unit resource systems that uses an adjacency matrix representation and graph reduction. When implemented in hardware, its overall run-time complexity is only $O(\min(m,n))$. For multi-unit resource systems, Shoshani et al. [10] introduced an algorithm with an $O(m^2 \times n)$ run-time complexity, leveraging a Resource Allocation Graph (RAG). Based on the same RAG representation, Holt [3] devised a deadlock detection algorithm with a reduced $O(m \times n)$ run-time complexity for multi-unit resource systems. Leibfried [7] introduced an adjacency matrix representation for resource allocations and performed deadlock detection in the means of matrix multiplication, which has an $O(m^2)$ run-time complexity. A decade ago, having extended his previous work [5], Kim presented an algorithm that detects deadlock in $O(1)$ run-time for multi-unit resource systems [4], but its overall run-time remains $O(m \times n)$, which has not been improved since Holt’s work [3]. Compared with all previously published algorithms, our parallel deadlock detection algorithm is the first that is applicable to multi-unit resource systems with only an $O(\min(m,n))$ overall run-time complexity.
3 A New O(1) Deadlock Detection Methodology for Multi-unit Resource Systems

In this section, we first present some assumptions for our target system. Then, we describe a matrix representation [9] of a RAG, on which our hardware algorithm is based. Next, the details of our algorithm for multi-unit resource systems are presented with an example.

3.1 Assumptions

For our deadlock detection algorithm, we make the following assumptions:

1. The operating system grants resources immediately if the requested resource units are available, which makes the entire system expedient [3].

2. A process requests one resource unit at a time. Thus, a process is blocked as soon as it requests an unavailable resource. As a result, a knot becomes a necessary and sufficient condition for deadlock [3].

See next paragraph for the definition of a knot. Note that relaxing Assumption 2 is our future work.

3.2 Introduction of A Matrix Representation of A Weighted RAG

Resource allocation among processes and resources in a system can be represented by a Resource Allocation Graph (RAG). A RAG is defined as a graph \((V, E)\) where \(V\) is a set of nodes and \(E\) is a set of directed edges. \(V\) can be further partitioned into two disjoint subsets: the process set \(P\) and the resource set \(Q\). A RAG is a bipartite graph in these two sets. An edge \(e_{ij} = (p_i, q_j)\) is a request edge if and only if \(p_i \in P\) and \(q_j \in Q\). An edge \(e_{ji} = (q_j, p_i)\) is a grant edge if and only if \(q_j \in Q\) and \(p_i \in P\). A node is a sink if and only if the node does not have any outgoing edge. A path is a sequence of alternating nodes and edges \((p_{i_1}, q_{j_1}), (q_{j_1}, p_{i_2}), \ldots, (p_{i_k}, q_{j_k}), \ldots, (q_{j_k}, p_{i_{k+1}})\), where each edge is distinct. If any nodes in a path are not distinct, it contains at least one cycle. A cycle does not contain any sink. The reachable set of node \(a\) is the set of nodes such that a path exists from \(a\) to every node in the reachable set. A knot is a nonempty set \(K\) of nodes such that the reachable set of each node in \(K\) is exactly \(K\) [3].

In [9], the authors present a RAG using an adjacency matrix. Each element of such a matrix represents either a request edge (denoted by \(r\)), grant edge (denoted by \(g\)) or no edge (denoted by \(0\)) in a RAG. Figure 1(a) shows an example of the adjacency matrix of a system with \(m\) processes \((p_1, p_2, \ldots, p_m)\) and \(n\) resources \((q_1, q_2, \ldots, q_n)\).

However, in a system with multi-unit resources, a process may request and thus is assigned with not only multiple resources but also more than one unit of the same type of resource. Such a case is represented in the RAG by multiple edges each with an integer denoting the number of units assigned. We call this alternative representation a weighted RAG. To properly represent such a weighted RAG, in addition to the adjacency matrix, we use another \(m\) by \(n\) matrix, which we call RAG weight matrix \(W\) or weight matrix in short. Each entry of \(W\) stores the number of units of a resource that are assigned to each process. Furthermore, we deploy a row vector of size \(n\) (called free-unit vector) to keep track of the numbers of available units of all resource types. Figure 1(b) shows an example of a weight matrix and its free-unit vector for a system with multi-unit resources.

In summary, a weight matrix for a weighted RAG \((V, E)\) can be defined as follows: \(W = [w_{ij}]_{m \times n}, (1 \leq i \leq m, 1 \leq j \leq n),\) and where

\[
 w_{ij} = t \text{ if } t \text{ units of } q_j \text{ are granted to } p_i.
\]

A free-unit vector can be defined as follows: \(F = [f_j]_n,\) \((1 \leq j \leq n),\) and where

\[
 f_j = k \text{ if } k \text{ units of } q_j \text{ are free (i.e., available)}.
\]

3.3 The Behind Theory

According to Assumption 2, for each process node in the RAG, there can exist at most one outstanding request edge. Therefore, each row in the adjacency matrix \(M\) can contain at most one \(r\) value. If a process has no pending request, then its node in the RAG has no outgoing edges, and its corresponding row in the matrix has no \(r\) values. Such a process node is called sink in graph theory [2, 3, 4]. A sink node represents an active process because the process has acquired all the necessary resources to run. As stated in Kim’s algorithm [4], if a resource is requested by a sink process that is the only reachable sink node for that resource node in the RAG, a deadlock occurs. Thus, it is observed that as long as we have associated each resource type with its proper sink nodes, we can detect deadlock in \(O(1)\) time [4]. Note that a multiple unit resource can have multiple sinks.
However, unlike Kim’s algorithm, we compute the associated sink nodes for each and every resource type in parallel using the adjacency matrix. To compute sink nodes for each resource, all paths that start at the resource are tracked in parallel until the entire reachable set of this resource has been discovered. Due to the fact that the RAG is a bipartite graph, process and resource nodes are alternated along any path. Thus, the overall complexity of computing the sink nodes for each resource is bounded by the length of the longest path in the RAG, which is $O(\min(m, n))$ [6].

It is important to note that we use the adjacency matrix not the weight matrix in the computation of sink nodes, the reason of which can be explained with the following theorem:

**Theorem 3.1** An expedient general resource graph with single-unit requests is a deadlock state if and only if it contains a knot [3].

According to Theorem 3.1, if we find a knot in an expedient RAG with single-unit requests (Assumptions 1 and 2), the system is in a deadlock state. Also, by the definition of a knot [3], a knot is formed when an exclusive set of nodes can only reach the nodes in the same set. Note here that this definition only considers whether a path exists between two nodes, but does not count weights that are assigned to the edges along that path. Therefore, it is enough to consider only the adjacency matrix for deadlock detection, which contains all edges in the RAG.

On the contrary, the RAG weight matrix and the free-unit vector are utilized to track the status of resource allocation inside the system. When a unit of any type of resource is granted, the corresponding entry is increased in the RAG weight matrix and decreased in the free-unit vector. On the other hand, when a unit of a resource type is released, the corresponding entry is decreased in the RAG weight matrix and increased in the free-unit vector. The released resource unit becomes available for a future assignment.

### 3.4 Our Algorithm and Its Detail

Although our description is well self-contained, since our work extends both Kim’s [4] and Shiu’s [9] work, the readers are suggested to read their work to have some background if necessary. Before we present the parallel Multi-unit resource Deadlock Detection Algorithm (MDDA), let us first introduce and explain some data structures used in the algorithm as shown in Table 1. In this paper, matrix[], matrix[i][] and matrix[i][][] refer to as “all elements in the matrix,” “all elements of row i in the matrix,” and “all elements of column j in the matrix,” respectively.

For the sake of our parallel algorithm, an adjacency matrix for a RAG$(V, E)$ is considered separately in two matrices without loss of generality: one containing only grant edges and the other containing only request edges. In summary, a grant adjacency matrix ($MG$) can be defined as follows: $MG = [m_{ij}]_{m \times n}, (1 \leq i \leq m, 1 \leq j \leq n),$

$$m_{ij} = \begin{cases} 1 & \text{if } \exists(q_i, p_j) \in E, \\ 0 & \text{otherwise}. \end{cases}$$

A request adjacency matrix ($MR$) can be defined as follows: $MR = [m_{ij}]_{m \times n}, (1 \leq i \leq m, 1 \leq j \leq n),$

$$m_{ij} = \begin{cases} 1 & \text{if } \exists(p_i, q_j) \in E, \\ 0 & \text{otherwise}. \end{cases}$$

A RAG weight matrix ($W$) and a free-unit vector ($WorkResource[]$) have been defined in Section 3.2. FreeUnit[] allows the algorithm to decide whether a request can be granted or should be blocked.

A sink process vector allows the algorithm to immediately know whether or not a reached process node is a sink node during the search for sink nodes. A sink process vector can be defined as follows: $SP = [sp_i]_m, (1 \leq i \leq m),$

$$sp_i = \begin{cases} 1 & \text{if } p_i \text{ is sink}, \\ 0 & \text{otherwise}. \end{cases}$$

<table>
<thead>
<tr>
<th>name</th>
<th>notation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGrant[i][j]</td>
<td>$mg_{ij}$</td>
<td>whether any units of resource $j$</td>
</tr>
<tr>
<td>MRequest[i][j]</td>
<td>$mr_{ij}$</td>
<td>whether any request for resource $j$</td>
</tr>
<tr>
<td>Weights[i][j]</td>
<td>$w_{ij}$</td>
<td>the number of units of resource $j$</td>
</tr>
<tr>
<td>FreeUnit[i][j]</td>
<td>$f_j$</td>
<td>the number of free units of resource $j$</td>
</tr>
</tbody>
</table>

The matrices mentioned so far maintain their states always whereas the following four matrices are initialized every time Step 4 of Algorithm 3.2 is entered.

A sink-ID matrix stores information about the reachable sink nodes for each and every resource and is used to detect deadlock in a constant run-time. A sink-ID matrix ($Sink[]$) can be defined as follows: $SK = [sk_{ij}]_{m \times n}, (1 \leq i \leq m)$
We call the column vector, Sink\[j\], the "bitmask" for the sink nodes of \(q_j\).

WorkProcess[] and WorkResource[] are used to store the process and resource nodes visited during the intermediate steps of computing sink bitmasks, respectively. A work process matrix (WorkProcess[]) can be defined as follows: 
\[
WP = [wp_{ij}]_{m \times n}, (1 \leq i \leq m, 1 \leq j \leq n),
\]

\[
wp_{ij} = \begin{cases} 
1 & \text{if the latest process visited by the search started from } q_j \text{ is } p_i, \\
0 & \text{otherwise.}
\end{cases}
\]

A work resource matrix (WorkResource[]) can be defined as follows: 
\[
WR = [wr_{jk}]_{n \times n}, (1 \leq j \leq n, 1 \leq k \leq n),
\]

\[
wr_{jk} = \begin{cases} 
1 & \text{if the latest resource visited by the search started from } q_j \text{ is } q_k, \\
0 & \text{otherwise.}
\end{cases}
\]

PrevResource[] is used to prevent the algorithm from visiting the same resource node more than once, ensuring \(O(min(m, n))\). A matrix (PrevResource[]) for previously-visited resource nodes can be defined as follows: 
\[
PR = [pr_{jk}]_{n \times n}, (1 \leq j \leq n, 1 \leq k \leq n),
\]

\[
pr_{jk} = \begin{cases} 
0 & \text{if resource } q_k \text{ has been visited during the search for } q_j \text{’s sink nodes,} \\
1 & \text{otherwise.}
\end{cases}
\]

Lastly, we define the bitmask of \(p_i\) as an \(m\)-digit vector 
\[
[0 \ldots 010 \ldots 0]_T
\]
where only the \(i^{th}\) bit is one.

**Algorithm 3.2 Parallel O(1) Multi-unit Resource Deadlock Detection Algorithm**

**Step 1: Resource request event\((p_i, q_j)\)**
1. when \(p_i\) makes a request for a unit of \(q_j\), do
2. if FreeUnit[][] > 0
3. grant a unit of \(q_j\) to \(p_i\)
4. MGrant[][i][j] = 1 // no deadlock results
5. increase Weight[][i][j] by one
6. decrease FreeUnit[][] by one
7. go to Step 4
8. else
9. MRequest[][i][j] = 1 // deadlock may exist
10. go to Step 3

**Step 2: Resource release event\((p_i, q_j)\)**
11. when \(p_i\) releases a unit of \(q_j\), do
12. update MGrant[], MRequest[], Weights[] and FreeUnit[]
13. if any process (say \(p_s\)) has been blocked for \(q_j\)
14. Resource request event\((p_s, q_j)\)
15. else go to Step 4

**Step 3: Detect Deadlock\((p_i, q_j)\)**
16. if (the bitmask of \(p_i\) == Sink[][j]) (i.e. the bitmask for \(q_j\’s\) sink nodes)
17. // \(p_i\) is the only sink node for \(q_j\)
18. deadlock exists
19. else
20. no deadlock exists
21. go to Step 4

**Step 4: Update sink bitmasks**
22. \(\forall i = 1, \ldots, m\), SinkProcess[i] = \(\sim(\lor_{1 \leq j \leq n} MRequest[i][j])\)
23. for all \(q_j (j = 1, \ldots, n)\) // do in parallel
24. Sink[][j] = \([0 \ldots 0]_{m \times 1}\), WorkProcess[][j] = \([0 \ldots 0]_{m \times 1}\)
25. if MGrant[][j] contains at least one non-zero element
26. \(\forall k = 1, \ldots, n\), WorkResource[][k][j] = 1 if \(k = j\); 0 otherwise
27. \(\forall k = 1, \ldots, n\), PrevResource[][k][j] = 0 if \(k = j\); 1 otherwise
28. update MGrantigate (j), MRequestigate (j) and Weightsigate (j)
29. while(1) // track \(q_j\’s\) sink ids
30. \(\forall i = 1, \ldots, m\), WorkProcess[i][j][j]
31. if WorkResource[][j][j] contains all zeros
32. stop for \(q_j\)

* Lines 27 to 30 are computed in sequence.
* \(\sim, \lor, \land\) denote NOT, bit-wise OR and AND, respectively.

In the rest of this section, we illustrate the operation of our algorithm with a system example consisting of five processors and four multi-unit resources, as shown in Figure 2.

Note that one process is running on each processor. Figure 2(b) shows a current resource allocation status in a form of weighted RAG. The corresponding adjacency matrix representation of the status is shown in Table 2. Note that the number of total units for each resource is specified in Figure 2(a).

**Figure 2. SoC Example**

We will show how sink node bitmasks are computed for each resource in Step 4. The resulting sink node bitmasks will be saved in Sink[] and then used for future deadlock detection in Step 3. Our algorithm stores all the reachable sink nodes for resource node \(j\) in Sink[][j] (i.e., \(j^{th}\) column of Sink[]). Next time, when a request for resource \(j\) is blocked (line 9), the algorithm can immediately detect
deadlock by comparing the requester process bitmap with the saved \( sink \) bitmask in Sink[1][j] (line 15).

The following matrices show some data structures (i.e., MGrant[], MRequest[], Weights[], FreeUnit[] and SinkProcess[]) that were computed at the last invocation of the algorithm (the abbreviated notations in Table 1 are used):

\[
\begin{align*}
M_{G_{m\times n}} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} & MR_{m\times n} &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\
W_{m\times n} &= \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} & F_n &= \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} & SP_m &= \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \end{bmatrix}^T
\end{align*}
\]

\( M_{G_{m\times n}} \) stores the same grant edges from resources to processes as in Table 2. \( MR_{m\times n} \) stores the same request edges from processes to resources as in Table 2. \( W_{m\times n} \) stores the numbers of units of each resource assigned to each process. \( F_n \) shows that all units of every resource have been granted; thus, there exists no available resource units in the current system. \( SP_m \) indicates that in the current RAG, processes \( p_2 \) and \( p_3 \) are \( sink \) nodes. Note that \( SP_m \) is computed from \( MR_{m\times n} \) using the following equation (line 20):

\[
\forall i \, sp_i = \sim (\vee_{1 \leq t \leq m} mr_{it}i).
\]

Right before the iteration, several data structures (i.e., Sink[], WorkProcess[], WorkResource[] and PrevResource[]) are initialized in lines 22-25. That is, in our example all elements in Sink[] and WorkProcess[] are initialized to zeros (SK, WP = [0]_{m\times n}). WorkResource[] and PrevResource[] are initialized as follows:

\[
WR_{n\times m} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & PR_{n\times m} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}
\]

The current bit patterns of \( WR \) and \( PR \) imply that the search starts from each resource (and visited each resource itself). Now the journey to search for \( sink \) nodes begins. The algorithm goes into the iterations of a loop of computing \( sink \) node bitmasks and collects them in Sink[].

First, the algorithm computes WorkProcess[] (i.e., currently visiting processes) from WorkResource[] (i.e., most recently visited resources) using MGrant[] (line 27). The operation in line 27 is equivalent to the following equation:

\[
\forall i, j \, wp_{ij} = \forall_{1 \leq t \leq n} (\text{wr}_{jt} \land \text{mg}_{it}).
\]

The computations of \( wp_{ij} \) through \( wp_{m,j} \) are equivalent to finding all processes that hold some units of resource \( q_i \) denoted in WorkResource[1][] (i.e., \( j^{th} \) row) and storing them in WorkProcess[1][]). Now let us choose resource \( q_1 \) of Figure 2 as an example to discover its \( sink \) nodes and take a closer look at how the matrices are computed. In the RAG of Figure 2(b), the units of \( q_1 \) are assigned to three processes \( (p_1, p_2 \) and \( p_3 \). At the same time, \( p_5 \)'s request for \( q_1 \) is blocked. Accordingly, MGrant[1][] is \([11100]^T\) (ones at the 1st, 2nd and 3rd elements which correspond to \( g \) values in the adjacency matrix), and MRequest[1][] is \([00001]^T\) (1 at the 5th element which corresponds to \( r \) value in the adjacency matrix). Before the first iteration, WorkResource[1][] was initialized as \([1000]^T\) (line 24). Thus, WorkProcess[1][] is first computed and stored as the bitmask \([11100]^T\) (line 27) since \( p_1, p_2 \) and \( p_3 \) are directly connected from \( q_1 \). In the same way, for the rest of resources, their corresponding columns in WorkProcess[] (WorkProcess[2][] through WorkProcess[n][])) are computed. The following values of WorkProcess[] result:

\[
WP_{m\times n} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\]

Then, if any process marked in WorkProcess[][] is a \( sink \) node, it is inserted into Sink[][] (line 28). The operation in line 28 is equivalent to the following equation:

\[
\forall i, j \, sk_{ij} = (wp_{ij} \land sp_i) \vee sk_{ij}.
\]

For the case of \( q_1 \), the algorithm checks whether anyone is a \( sink \) node among the three process nodes \( (p_1, p_2 \) and \( p_3 \) by comparing WorkProcess[1][1] \([11100]^T\) with SinkProcess \([01100]^T\) (line 28). As a result, \( p_2 \) and \( p_3 \) are identified as newly found \( sink \) nodes for resource \( q_1 \). Thus, they are saved in Sink[1][] (the 2nd and 3rd elements of the 1st column become ones). Similarly, the rest of Sink[] (Sink[2][] through Sink[n][])) are computed for resource 2 through resource \( n \). The following values of Sink[] result:

\[
SK_{m\times n} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
\]

Next, the algorithm computes WorkResource[] from WorkProcess[] (just computed), MRequest[] and PrevResource[]. Doing that, the algorithm finds a set of resource nodes that are directly connected from processes noted.
in WorkProcess[i][j] for each resource \( j \) and stores them in WorkResource[i][j] (line 29). This operation is equivalent to traveling one step farther from the originated resource node to another resource node via a process blocked for the second resource node. The operation in line 29 can be expressed by the following equation:

\[
\forall j, k \ wr_{jk} = \vee_{1 \leq c \leq m} (wp_{cj} \land mr_{ck}) \land pr_{jk}.
\]

To illustrate the computation of WorkResource[, let us continue with the example for \( q_1 \). WorkProcess[][1] \([11100]^T\) indicates that \( q_1 \) now reaches three process nodes \((p_1, p_2\) and \( p_3 \)). From these three process nodes, \( p_1 \) has one outgoing edge (a request edge) to \( q_2 \); \( p_2 \) and \( p_3 \) have no outgoing edges since they are \( sink \) nodes. Thus, algorithm computes WorkResource[1][] as \([0100]\) (line 29), indicating that \( q_1 \) can further reach resource node \( q_2 \) via \( p_1 \) since \( p_1 \) is blocked for \( q_2 \). In the same way, the rest of WorkResource[] (WorkResource[2][] through WorkResource[n][]) are computed for resource 2 through resource \( n \). The following values of WorkResource[] result:

\[
WR_{n \times n} = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\]

This \( WR \) implies the following as can be seen from Figure 2(b): \( q_1 \) can reach \( q_2 \) via \( p_1 \), \( q_2 \) can reach \( q_1 \) and \( q_3 \) via \( p_2 \) and \( p_3 \), respectively. \( q_3 \) and \( q_4 \) have no paths to reach other resources.

After WorkResource[] is computed, the algorithm updates PrevResource[] to include any resource node that is marked as 1 (i.e., just visited) in WorkResource[] (line 30). The operation in line 30 can be expressed by the following equation:

\[
\forall j, k \ pr_{jk} = \sim wr_{jk} \land pr_{jk}.
\]

Continue with the example of \( q_1 \). PrevResource[1][] was initialized as \([0111]\) (line 25). Currently, the values of WorkResource[1][] are \([0100]\), meaning the search for \( q_1 \)'s \( sink \) nodes is now visiting \( q_2 \). Thus, the updated PrevResource[1][] is computed as \([0011]\) (line 30), which signifies both \( q_1 \) and \( q_2 \) have now been visited (0 indicates “visited”). This mechanism effectively prevents our algorithm from being trapped in cycles. The rest of PrevResource[] (PrevResource[2][] through PrevResource[n][]) are computed for resource 2 through resource \( n \) in the same way. The following values of PrevResource[] result:

\[
PR_{n \times n} = \begin{bmatrix}
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 
\end{bmatrix}
\]

This matrix implies that starting from \( q_1 \), the path has currently reached \( q_2 \); from \( q_2 \) reaching \( q_1 \) as well as \( q_3 \).

After the computations from line 27 through line 30 are finished, the algorithm looks at each row of WorkResource[] (corresponding to each resource) to decide whether it can stop the search for sink nodes for the resource (line 31-32). If WorkResource[i][] contains all zeros, \( q_j \)'s search for its sink nodes has visited all resource nodes in \( q_j \)'s reachable set. Therefore, Sink[]= contains a complete bitmask for \( q_j \)'s reachable sink nodes. Thus, \( q_j \)'s search can be stopped after the current iteration. Note that searches for different resource nodes may stop at different iterations. As in this example, after the first iteration is finished in Step 4, for \( q_3 \) and \( q_4 \), their corresponding rows in matrix \( WR \) (the \( 3^{rd} \) and \( 4^{th} \) rows) contain all zero values. This means the current values for \( q_3 \) and \( q_4 \) in matrix \( SK \) (the \( 3^{rd} \) and \( 4^{th} \) columns) contain their complete sink node bitmasks. This informs that \( q_3 \) and \( q_4 \) have their sink node bitmasks equal to \( p_2 \) and \( p_3 \), respectively. Thus, for \( q_3 \) and \( q_4 \), there is no need to perform the search further. For \( q_1 \), however, the \( 1^{st} \) row \(([0100])\) of matrix \( WR \) contains one element whose value is 1 (the \( 2^{nd} \) element in the row), meaning that some of the three processes, \( p_1 \), \( p_2 \) and \( p_3 \) (which are stored as bitmask \([11100]^T\) in WorkProcess[][1]), are blocked for resource \( q_2 \). Similarly for \( q_2 \), the \( 2^{nd} \) row \(([1010])\) of matrix \( WR \) indicates that either or both of the two processes, \( p_4 \) and \( p_5 \) \(([00011]^T\) as noted in WorkProcess[][2]), are blocked for resources \( q_1 \) and \( q_3 \). Thus, for \( q_1 \) and \( q_2 \), further iterations must be performed to track and reach more sinks.

In the second iteration, all contents corresponding to \( q_1 \) and \( q_2 \) are computed in the same way as in the first iteration whereas contents corresponding to \( q_3 \) and \( q_4 \) remain unchanged. The following results after the second iteration:

\[
WP_{n \times n} = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 
\end{bmatrix}
\]

\[
SK_{n \times n} = \begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\]

\[
WR_{n \times n} = \begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\]

\[
PR_{n \times n} = \begin{bmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 
\end{bmatrix}
\]

Now for \( q_2 \), the \( 2^{nd} \) row of \( WR \) also contains all zeros. It means that \( q_2 \)'s column in \( SK \) (the \( 2^{nd} \) column) contains its complete sink bitmask. For \( q_1 \), however, its corresponding row of matrix \( WR \) (the \( 1^{st} \) row) still contains an element whose value is 1. Thus, further iterations must be performed to track more sinks. After another iteration of computation is performed for \( q_1 \), the following results:

\[
WP_{n \times n} = \begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\]

\[
SK_{n \times n} = \begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\]
Now for \( q_1 \), its row (the 1st row) of \( WR \) also contains all zeros. It means that its column in \( SK \) (the 1st column) contains its complete \( sink \) bitmask. From matrix \( SK \), we can see that \( q_1 \) and \( q_2 \) are bonded to two \( sink \) nodes \( p_2 \) and \( p_1 \), while \( q_3 \) and \( q_4 \) are bonded to their single \( sink \) nodes \( p_2 \) and \( p_3 \), respectively. Note that the search for each resource node’s \( sinks \) only visits the resource nodes in its reachable set once. This guarantees that the complexity of updating all \( sink \) bitmasks is bounded by \( O(\min(m, n)) \).

Let us now suppose that from the current state of Figure 2(b), \( p_2 \) makes a request for \( q_4 \) and nothing else changes in the system. Does this request cause deadlock? Since \( q_4 \) has only one unit being used by \( p_3 \), this request is suspended. Thus, \( p_2 \) must wait for \( q_4 \) to become free. Then, the deadlock detection unit is invoked immediately (Step 3).

As indicated in our algorithm, we simply compare \( q_4 \)’s \( sink \) bitmask \( Sink[][4] \) (\( [00100]^T \)) with the bitmask of \( p_2 \) (\( [01000]^T \)). Since they are different, we can conclude that this request does not cause deadlock in the system. Accordingly, all \( sink \) bitmasks for all resource nodes are updated in Step 4. After the updating, all resource nodes are bonded to one single \( sink \) node \( p_3 \).

Furthermore, after \( p_2 \) is blocked for \( q_4 \), let us assume \( p_3 \) makes a request for a unit of \( q_2 \). Does this request cause deadlock? \( p_3 \) is blocked for this request because \( q_2 \) has two units being used by \( p_4 \) and \( p_2 \). By comparing \( q_2 \)’s \( sink \) node bitmask \( Sink[][2] \) (\( [00100]^T \)) with the bitmask of \( p_1 \) (\( [00100]^T \)), the algorithm immediately determines that the request causes deadlock.

### 3.5 Architecture

In this section, we delineate how Algorithm 3.2 is realized in hardware. The simplified view of the architecture of a Multi-unit resource hardware Deadlock Detection Unit (MDDU) for 3 processes and 3 resources is shown in Figure 3. We show 3x3 MDDU instead of 5x4, which might look too messy given the space of the paper. The architecture implements all data structures in Table 1. In Figure 3, each Matrix cell contains three elements one each from MGrant[], MRequest[] and Weights[]. Each SK/WP cell contains two elements one each from Sink[] and WorkProcess[]. Each WR/PR cell contains two elements one each from WorkResource[] and PrevResource[]. The computations in Step 4 of Algorithm 3.2 are performed by acting on the data structures maintained within the various cells (i.e., Matrix cells, SK/WP cells and WR/PR cells).

We used the Xilinx ISE 9.1i to synthesize 5x5 MDDU on Virtex-II XC2V250 device. MDDU utilizes 757 slices (two slices form one Configurable Logic Block (CLB)). In total, 650 four-input Look-Up Tables (LUTs) were utilized. The total equivalent gate count for our design is 8949.

### 4 Experiment and Discussion

#### 4.1 Target System and Simulation Environment

Here we describe a sample target Multiprocessor System-on-a-Chip (MPSoC) integrating our 5x4 MDDU. The architecture of the target SoC is similar to Figure 2(a). It consists of five Motorola MPC755 processors, SRAM memory of 256MB and four multi-unit dummy resources. These dummy resources count the specified time (set by the software) after their units are granted and interrupt the corresponding processor where a process uses the resource units when a preset time elapses. Thus, it is sufficient to use these dummy devices to simulate various deadlock scenarios in the system. All components except processors and SRAMs are implemented in Verilog HDL. The application is written in the C language and compiled using a PowerPC-GCC cross-compiler. We use Atalanta RTOS version 0.3 [11], a shared memory multiprocessor RTOS, to manage processes and resources. In order to simulate such an MPSoC with the multiprocessor operating system, we use Mentor Graphics Seamless hardware-software Co-Verification Environment [8] aided by ModelSim for hardware simulation and XRAY for software debugging.

#### 4.2 Application Scenario

We test the 5x4 MDDU with an application consisting of five processes each running on a processor of the aforementioned target. Due to space limit, we are unable to describe
the scenario in detail, which has been embedded in Section 3.4. Instead, we provide a technical report [12] for the complete explanation of the application.

4.3 Experimental Results

We simulated a series of events including both grants and requests, which incrementally builds up the resource allocation situation shown in Figure 2(b). Throughout our simulation, deadlock detection always takes only two clock cycles for all events. To demonstrate the reduced detection preparation time, we count the number of clock cycles between the moment the detection preparation is triggered and the moment all W R elements become zeros in ModelSim's wave window. The events and their corresponding detection preparation times are listed in Table 3. Note that all grants and requests are made for one unit per event according to Assumption 2. As shown in the table, MDDU takes 15 clock cycles to compute the sink bitmasks for all resources in the RAG shown in Figure 2(b) at time . That is equivalent to three iterations of the while loop in Step 4. In our simulated target, the maximum number of iterations in Step 4 can be four at most because of the aforesaid \( O(\min(m, n)) \) run-time complexity where \( m = 5 \) and \( n = 4 \). Finally, at time \( t_{13} \), deadlock is detected in the system.

Table 3. Simulated Events and Their Detection Preparation Clock Cycles on 5x4 Multi-unit Resource MPSoC

<table>
<thead>
<tr>
<th>Time</th>
<th>Events</th>
<th>Preparation</th>
<th>Time</th>
<th>Events</th>
<th>Preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>( q_1 ) is granted to ( p_1 )</td>
<td>5</td>
<td>( t_6 )</td>
<td>( q_1 ) is granted to ( p_2 )</td>
<td>10</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( q_1 ) is granted to ( p_2 )</td>
<td>5</td>
<td>( t_9 )</td>
<td>( q_1 ) is granted to ( p_3 )</td>
<td>10</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>( q_2 ) is granted to ( p_1 )</td>
<td>5</td>
<td>( t_{10} )</td>
<td>( p_2 ) requests ( q_3 )</td>
<td>15</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( q_2 ) is granted to ( p_2 )</td>
<td>5</td>
<td>( t_{11} )</td>
<td>( p_2 ) requests ( q_1 )</td>
<td>15</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>( q_2 ) is granted to ( p_4 )</td>
<td>5</td>
<td>( t_{12} )</td>
<td>( p_2 ) requests ( q_4 )</td>
<td>15</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>( p_2 ) requests ( q_1 )</td>
<td>10</td>
<td>( t_{13} )</td>
<td>( p_3 ) requests ( q_2 )</td>
<td>N/A</td>
</tr>
<tr>
<td>( t_7 )</td>
<td>( q_1 ) is granted to ( p_1 )</td>
<td>10</td>
<td>( t_{14} )</td>
<td>( p_3 ) requests ( q_1 )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

4.4 Discussion

As both of ours and Shiu’s algorithms have the same \( O(\min(m, n)) \) overall run-time complexity, let us compare various aspects of our algorithm with those of Shiu’s algorithm, and highlight their difference and our novelty. A summary of the comparison is shown in Table 4. As can be seen in the table, other than the same overall run-time complexity and the single-request assumption for processes, the two algorithms are different in all aspects.

5 Conclusion

An \( O(1) \) parallel multi-unit resource deadlock detection algorithm is presented in this paper. The described algorithm has a run-time complexity of \( O(1) \) for detecting deadlock and \( O(\min(m, n)) \) for preparing detection. Compared with Kim’s algorithm [4], deadlock preparation run-time complexity is significantly reduced from \( O(m \times n) \) to \( O(\min(m, n)) \). For current and future SoCs, our MDDU provides very fast and deterministic run-time detection of deadlocks for both multi-unit as well as single-unit resource systems.

References