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 $\mathcal{O}(1)$ parallel deadlock detection approach for multi-unit resource System-on-a-Chips (SoCs), inspired by Kim's method in $\mathcal{O}(1)$ detection as well as Shiu's method in parallel processing. Our contributions are (i) the first $\mathcal{O}(1)$ hardware deadlock detection and (ii) $\mathcal{O}(min(m,n))$ preparation, both for multi-unit resource systems, where m and n are the number of processes and resources, respectively. $\mathcal{O}(min(m,n))$, previously $\mathcal{O}(m \times n)$, 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 $\mathcal{O}(1)$ multi-unit resource deadlock detection and (ii) a reduced $\mathcal{O}(min(m,n))$ 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 $\mathcal{O}(m \times n)$ [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 $\mathcal{O}(1)$ deadlock detection run-time and an $\mathcal{O}(m+n)$ 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 $\mathcal{O}(min(m,n))$. For multi-unit resource systems, Shoshani *et al.* [10] introduced an algorithm with an $\mathcal{O}(m^2 \times n)$ 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 $\mathcal{O}(m \times n)$ 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 $\mathcal{O}(m^3)$ run-time complexity. A decade ago, having extended his previous work [5], Kim presented an algorithm that detects deadlock in $\mathcal{O}(1)$ run-time for multi-unit resource systems [4], but its overall run-time remains $\mathcal{O}(m \times n)$, 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 $\mathcal{O}(min(m,n))$ 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}), $\ldots$, $(q_{i_s}, p_{j_{s+1}})$, 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 represent 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 $(p_1, p_2, ..., p_m)$ and n resources $(q_1, q_2, ..., q_n)$.

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.

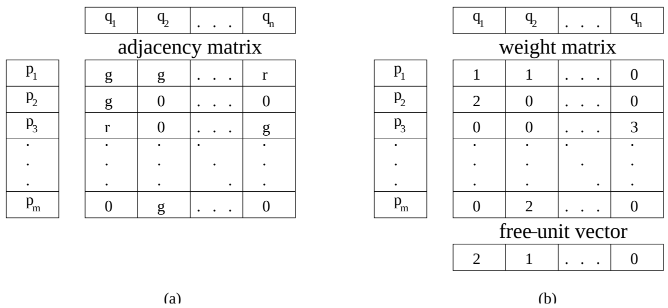

Figure 1. An example RAG adjacency matrix, its weight matrix and free-unit vector.

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

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

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

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

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 $\mathcal{O}(1)$ 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 $\mathcal{O}(min(m,n))$ [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[][j] 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

Table 1. Data structures for MDDA.

name	notation	description
MGrant[i][j] in $MG_{m\times n}$	mg_{ij}	whether any units of resource j are granted to process i
MRequest[i][j] in $MR_{m\times n}$	mr_{ij}	whether any request for resource j from process i is blocked
Weights[i][j] in $W_{m\times n}$	w_{ij}	the number of units of resource j assigned to process i
FreeUnit[j] in F_n	f_j	the number of free units of resource j
SinkProcess[i] in SP_m	sp_i	whether process i is a $sink$ node in the RAG
Sink[i][j] in $SK_{m\times n}$	sk_{ij}	whether process i is a reachable $sink$ node in the RAG from resource j
WorkProcess[i][j] in $WP_{m\times n}$	wp_{ij}	whether process i is being visited in the current step during the search for resource j's $sink$ nodes
WorkResource[j][k] in $WR_{n\times n}$	wr_{jk}	whether resource k is being visited in the current step during the search for resource j's $sink$ nodes
PrevResource[j][k] in $PR_{n\times n}$	pr_{jk}	whether resource k has been visited in any previous steps during the search for resource j's $sink$ nodes

edges and the other containing only request edges. In summary, a grant adjacency matrix (MGrant[]) can be defined as follows: $MG = [mg_{ij}]_{m\times n}$, $(1 \le i \le m, 1 \le j \le n)$,

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

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

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

A RAG weight matrix (Weights[]) and a free-unit vector (FreeUnit[]) 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: $SP = [sp_i]_m$, $(1 \le i \le m)$,

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

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: $SK = [sk_{ij}]_{m\times n}$, $(1 \le i \le$

$m, 1 \leq j \leq n$),

$$sk_{ij} = \begin{cases} 1 & if\ p_i\ is\ sink\ and\ can\ be\ reached\ from\ q_j, \\ 0 & otherwise. \end{cases}$$

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: $WP = [wp_{ij}]_{m \times n}$, $(1 \leq i \leq m, 1 \leq j \leq n)$,

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

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

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

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

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

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[j] $> 0$
3            grant a unit of $q_j$ to $p_i$
4            MGrant[i][j] = 1    // no deadlock results
5            increase Weights[i][j] by one
6            decrease FreeUnit[j] 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[]
         if any process (say $p_s$) has been blocked for $q_j$
13           Resource request event($p_s, q_j$)
14       else go to Step 4

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

Step 4: Update sink bitmasks
20   $\forall$ i = 1,...,m, SinkProcess[i] =$\sim(\vee_{1 \leq t \leq n} MRequest[i][t])$
21   for all $q_j$ (j=1,...,n)   // do in parallel
22       Sink[][j] = $[0]_{m \times 1}$; WorkProcess[][j] = $[0]_{m \times 1}$
23       if MGrant[][j] contains at least one non-zero element
24           $\forall$ k = 1,...,n, WorkResource[j][k] = 1 if k==j; 0 otherwise
25           $\forall$ k = 1,...,n, PrevResource[j][k] = 0 if k==j; 1 otherwise
26           while(1)   // track $q_j$'s sink id's
27               $\forall$ i = 1,...,m, WorkProcess[i][j]
                     = $\vee_{1 \leq t \leq n}(WorkResource[j][t] \wedge MGrant[i][t])$
28               $\forall$ i = 1,...,m, Sink[i][j]
                     = $(WorkProcess[i][j] \wedge SinkProcess[i])$
                       $\vee Sink[i][j]$
29               $\forall$ k = 1,...,n, WorkResource[j][k]
                     = $\vee_{1 \leq c \leq m}(WorkProcess[c][j] \wedge MRequest[c][k])$
                     $\wedge PrevResource[j][k]$
30               $\forall$ k = 1,...,n, PrevResource[j][k]
                     = $\sim WorkResource[j][k] \wedge PrevResource[j][k]$
31               if WorkResource[j][] contains all zeros
32                   stop for $q_j$
```

* Lines 27 to 30 are computed in sequence.
* $\sim$, $\vee$ and $\wedge$ 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).

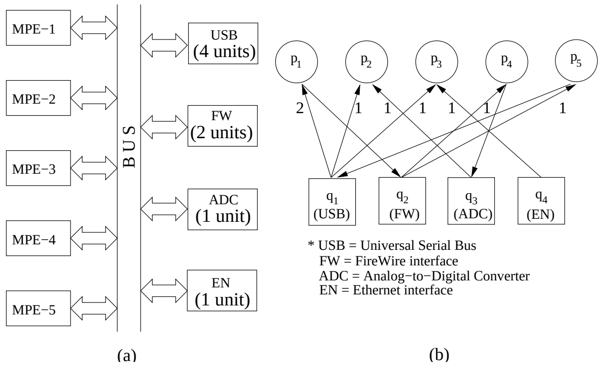

Figure 2. SoC Example

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

Table 2. The Adjacency Matrix for 5 Processes and 4 Resources

	q_1(USB)	q_2(FW)	q_3(ADC)	q_4(EN)
p_1	g	r	0	0
p_2	g	0	g	0
p_3	g	0	0	g
p_4	0	g	r	0
p_5	r	g	0	0

deadlock by comparing the requester process bitmask with the saved $sink$ bitmask in Sink[][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):

$$\mathbf{MG_{m\times n}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{MR_{m\times n}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{W_{m\times n}} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \begin{array}{l} \mathbf{F_n} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \\ \mathbf{SP_m} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \end{bmatrix}^T \end{array}$$

$MG_{m\times n}$ 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):

$$\forall i\ sp_i = \sim(\vee_{1\le t\le n} mr_{it}). \tag{1}$$

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 PrevResoruce[] are initialized as follows:

$$\mathbf{WR_{n\times n}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{PR_{n\times n}} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

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:

$$\forall i, j\ wp_{ij} = \vee_{1\le t\le n}(wr_{jt} \wedge mg_{it}). \tag{2}$$

The computations of wp_{1j} through wp_{mj} are equivalent to finding all processes that hold some units of resource q_j denoted in WorkResource[j][] (i.e., j^{th} row) and storing them in WorkProcess[][j]. 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 1^{st}, 2^{nd} and 3^{rd} elements which correspond to g values in the adjacency matrix), and MRequest[][1] is $[00001]^T$ (1 at the 5^{th} element which corresponds to r value in the adjacency matrix). Before the first iteration, WorkResource[1][] was initialized as $[1000]$ (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:

$$\mathbf{WP_{m\times n}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

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

$$\forall i, j\ sk_{ij} = (wp_{ij} \wedge sp_i) \vee sk_{ij}. \tag{3}$$

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] ($[11100]^T$) 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 2^{nd} and 3^{rd} elements of the 1^{st} 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:

$$\mathbf{SK_{m\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

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 notated in WorkProcess[][j] for each resource j and stores them in WorkResource[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:

$$\forall j,k\ wr_{jk} = \vee_{1\le c\le m}(wp_{cj} \wedge mr_{ck}) \wedge pr_{jk}. \quad (4)$$

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:

$$\mathbf{WR_{n\times n}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

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_5 and p_4, 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:

$$\forall j,k\ pr_{jk} = \sim wr_{jk} \wedge pr_{jk}. \quad (5)$$

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:

$$\mathbf{PR_{n\times n}} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

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[j][] contains all zeros, q_j's search for its $sink$ nodes has visited all resource nodes in q_j's reachable set. Therefore, Sink[][j] 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 notated 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:

$$\mathbf{WP_{m\times n}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{SK_{m\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{WR_{n\times n}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{PR_{n\times n}} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

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:

$$\mathbf{WP_{m\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{SK_{m\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{WR_{n\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{PR_{n\times n}} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Now for q_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_3, 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 $\mathcal{O}(min(m, n))$.

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_5. By comparing q_2's $sink$ node bitmask Sink[][2] ($[00100]^T$) with the bitmask of p_3 ($[00100]^T$), 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.

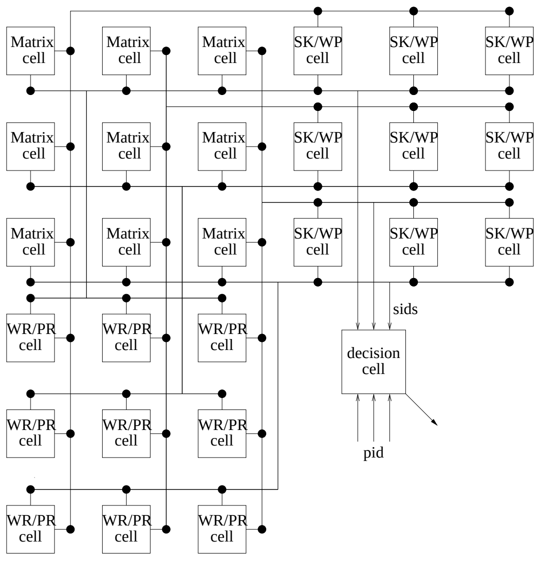

Figure 3. Architecture of 3x3 MDDU.

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 WR 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 t_{11}. 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 aforestated $\mathcal{O}(min(m,n))$ 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

Time	Events	Preparation	Time	Events	Preparation
t_1	q_1 is granted to p_1	5	t_8	q_1 is granted to p_2	10
t_2	q_3 is granted to p_2	5	t_9	q_1 is granted to p_3	10
t_3	q_4 is granted to p_3	5	t_{10}	p_4 requests q_3	15
t_4	q_2 is granted to p_5	5	t_{11}	p_5 requests q_1	15
t_5	q_2 is granted to p_4	5	t_{12}	p_2 requests q_4	15
t_6	p_1 requests q_2	10	t_{13}	p_3 requests q_2	N/A
t_7	q_1 is granted to p_1	10			

4.4 Discussion

As both of ours and Shiu's algorithms have the same $\mathcal{O}(min(m,n))$ 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 $\mathcal{O}(1)$ parallel multi-unit resource deadlock detection algorithm is presented in this paper. The described algorithm has a run-time complexity of $\mathcal{O}(1)$ for detecting deadlock and $\mathcal{O}(min(m,n))$ for preparing detection. Compared with Kim's algorithm [4], deadlock preparation run-time complexity is significantly reduced from $\mathcal{O}(m \times n)$ to $\mathcal{O}(min(m,n))$. 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.

Table 4. The Comparison between Our Algorithm and Shiu's Algorithm [9]

Feature	Ours	Shiu's
Applicable systems	Both multi-unit as well as single-unit resource systems.	Only single-unit resource systems.
Characteristics of the RAG	A resource node can have multiple outgoing edges (grants) and multiple incoming edges (requests).	A resource node can have at most one outgoing edge (grant) and multiple incoming edges (requests).
The theory behind the algorithm	A knot is a necessary and sufficient condition for deadlock.	A cycle is a necessary and sufficient condition for deadlock.
Key technique of the algorithm	Parallel graph traverse in search for sink process nodes.	Parallel graph reduction to remove terminal edges iteratively.
Detection complexity	$\mathcal{O}(1)$	$\mathcal{O}(min(m,n))$
Overall complexity	$\mathcal{O}(min(m,n))$	$\mathcal{O}(min(m,n))$

References

[1] International technology roadmap for semiconductors 2006 update. http://www.itrs.net, visited in May 2007.

[2] B. Claude. *The theory of graphs*. John Wiley & Sons, New York, 1962.

[3] R. Holt. "Some deadlock properties of computer systems," *ACM Computing Surveys*, 4(3):179–196, 1972.

[4] J. Kim. "Algorithmic approach on deadlock detection for enhanced parallelism in multiprocessing systems," In *PAS'97*, pages 233–238, 1997.

[5] J. Kim and K. Koh. "An O(1) time deadlock detection scheme in single unit and single request multiprocess system," In *IEEE TENCON '91*, pages 219–223, 1991.

[6] J. Lee and V. Mooney. "An O(min(m,n)) parallel deadlock detection algorithm," *ACM Trans. on Design Automation of Electronic Systems*, 10(3):573–586, 2005.

[7] T. Leibfried. "A deadlock detection and recovery algorithm using the formalism of a directed graph matrix," *Operating Systems Review*, 23(2):45–55, 1989.

[8] Mentor Graphics Corp. Hardware/software co-verification: Seamless. http://www.mentor.com/seamless, visited in May 2007.

[9] P. Shiu, Y. Tan and V. Mooney. "A novel parallel deadlock detection algorithm and architecture," In *CODES'01*, pages 73–78, 2001.

[10] A. Shoshani and E. Coffman. "Prevention, detection and recover from deadlock in multiprocess, multiple resource systems," Technical Report 80, Princeton University, 1969.

[11] D. Sun, D. Blough and V. Mooney. "Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications," Technical Report, GIT-CC-02-19, College of Computing, Georgia Tech, 2002.

[12] X. Xiao and J. Lee. "A novel parallel deadlock detection algorithm and hardware implementation for multi-unit resource systems," Technical Report, TR-ENGT-11, ECE Department, IUPUI, 2007.