A O(min(m,n)) Parallel Deadlock Detection Algorithm and Hardware for Multi-unit Resource Systems

Xiang Xiao and Jaehwan John Lee

ECE Department, Purdue School of Engineering and Technology

Indiana University-Purdue University Indianapolis

Abstract

This report describes a novel parallel Multi-unit resource Deadlock Detection Algorithm (MDDA) and its hardware implementation (MDDU). The contributions are (i) the first $\mathcal{O}(1)$ hardware deadlock detection, (ii) reduced $\mathcal{O}(min(m,n))$ preparation, where m and n are the number of processes and resources, respectively, and (iii) support for multi-unit resources. $\mathcal{O}(min(m,n))$, 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 two matrices representing resource allocations as well as other auxiliary matrices. Moreover, we prove the correctness and run-time complexity of MDDA. MDDU provides a fast and deterministic deadlock detection mechanism for Multiprocessor System-on-Chips (MPSoCs), which we predict will become prevalent in the near future in system designs. Our experiments demonstrate that MDDU always takes two clock cycles to detect deadlock regardless the size of the system. Lastly, the MPSoC area overhead due to MDDU is small, approximately 0.024 percent for MDDU16x16 on our example MPSoC.

Index Terms

Deadlock detection, Parallel algorithm, Multi-unit resource, Hardware/software codesign

I. Introduction

System-on-Chip (SoC) and Chip Multiprocessor (CMP) are new concepts featuring the design of single chips containing processor cores in addition to local memory and other Intellectual Property (IP)

cores. Such design paradigms are suitable for not only embedded systems but also high performance systems and provide the balance among cost, power consumption and performance. Recent technology enabling to integrate a tremendous number of transistors has allowed multiple processors to be fabricated on a single chip. In fact, the world is in an early stage of proliferation of CMPs for scientific and embedded computing domains as well as the general-purpose computing domain [2]. For example, the Cell Broadband Engine (CBE) is an 8-core heterogeneous CMP developed by Sony, Toshiba and IBM [6]. CBE-based embedded systems are envisioned as promising solutions for high-speed media processing systems and medical systems [7]. Another example is TILE64 processor family from Tilera [23]. One TILE64 processor consists of 64 identical processor cores (tiles) interconnected with Tilera's iMesh [24] on-chip network. Each tile integrates a full functional general purpose processor, L1 and L2 cache as well as a non-blocking switch that connects the tile into the mesh. As transistor counts continue to increase and in all likelihood follow Moore's Law [16], Multiprocessor System-on-Chips (MPSoCs) are very likely to become more prevalent in the near future.

To exploit the immense computing power of MPSoCs, multi-threaded applications or multi-program workloads likely become increasingly common in the future. If such a trend comes to be, MPSoCs will inevitably incur a high demand on resources as well. To avoid the resource demand becoming a huddle, it is common to provide multiple units of the same type of resource that can serve multiple processors simultaneously. While such MPSoCs can potentially improve the overall throughput of the systems, not only does the probability of deadlocks increase, but also deadlock detection becomes harder. As many MPSoCs are designed for real-time applications, they are managed by a Real-Time Operating System (RTOS). An RTOS must provide services that guarantee that deadlines be met deterministically. Due to these factors, the deadlock detection service for MPSoCs should be applicable to multi-unit resource multi-processor environment as well as fast and deterministic. To ensure that a deadlock detection service is delivered timely, it seems better to have a dedicated hardware unit that implements a deadlock detection algorithm because software deadlock detection would be less deterministic due to various execution overhead (e.g., OS context switch and resource contention).

In this report, we present a novel parallel Multi-unit resource Deadlock Detection Algorithm (MDDA) as well as its hardware implementation, which addresses the challenges for deadlock detection for the aforementioned real-time MPSoCs. Our work is inspired by Kim's work [9] in $\mathcal{O}(1)$ deadlock detection as well as Shiu's work [18] in parallel processing in hardware but makes substantial improvements in

various aspects. The described algorithm has a run-time complexity of $\mathcal{O}(1)$ for deadlock detection and $\mathcal{O}(min(m,n))$ for detection preparation, where m is the number of processes and n is the number of resources. $\mathcal{O}(1)$ complexity is achieved by pre-computing the reachable sink nodes of each resource and comparing the sink nodes with the requester process node for deadlock detection (see Definition 3.5 and 3.7 in Section III for the formal definitions of a reachable node and a sink node). The computation of the reachable sink nodes of a resource, in the phase of detection preparation, is equivalent to a graph traverse that visits all reachable nodes starting from that resource and then identifying sink process nodes. The graph traverse visits each resource node in its reachable set exactly once by our novel mechanism, which prevents our algorithm being trapped in cycles (see Definition 3.6 for the formal definition of a reachable set). Moreover, our algorithm parallelizes not only the graph traverse for multiple sinks of a resource but also graph traverses for each and every resource, thereby achieving $\mathcal{O}(min(m,n))$ complexity. We also prove the correctness and run-time complexity of MDDA.

The rest of this report is organized as follows: Section II reviews some related deadlock detection algorithms. Section III introduces definitions and terminologies relevant to MDDA. Section IV reveals the behind theory of MDDA. Section V elaborates on MDDA and explains the algorithm with an example. Section VI presents the proofs of the correctness and run-time complexity of MDDA. Section VII describes the hardware implementation of MDDA. Section VIII discusses the simulation of the MDDA hardware and the simulation results. Section IX concludes this report.

II. Related Work

Deadlock is a situation in which two or more processes are indefinitely waiting for resources that the other owns. A variety of deadlock detection algorithms have been proposed in the past. By the type of systems that an algorithm can be applied to, all previously published algorithms can be classified into two categories: the algorithms applicable to the systems with multi-unit resources and those applicable to the systems with only single-unit resources.

Shoshani *et al.* introduced a deadlock detection algorithm with a run-time complexity of $\mathcal{O}(m^2 \times n)$ for multi-unit resource systems, where m and n are equal to the number of processes and resources in the system, respectively [19]. Holt presented an $\mathcal{O}(m \times n)$ algorithm to determine whether a blocked process is deadlocked in multi-unit resource systems [5]. Both Shoshani's and Holt's algorithms leverage a Resource Allocation Graph (RAG) representation. Leibfried described a deadlock detection algorithm

for multi-unit resource systems using an adjacency matrix representation and matrix multiplication, but its run-time complexity is $\mathcal{O}(m^3)$ [12]. Kim also introduced an algorithm that detects deadlock in $\mathcal{O}(1)$ run-time for multi-unit resource systems [9]. However, the run-time complexity of its deadlock detection preparation is $\mathcal{O}(m \times n)$. For single-unit resource systems, Kim and Koh [10] introduced an algorithm that detects deadlock in $\mathcal{O}(1)$ run-time after the deadlock occurs and has the overall run-time complexity as $\mathcal{O}(m + n)$. Cahit [1] presented a deadlock detection method for single-unit resource systems based on the RAG, which has a run-time complexity of $\mathcal{O}(e)$, where e is the number of edges in the RAG. More recently, Shiu *et al.* [18] devised a parallel algorithm for single-unit resource systems which uses an adjacency matrix representation and graph reduction to detect deadlock, and its run-time complexity is only $\mathcal{O}(min(m, n))$. However, as most of today's SoCs contain multi-unit resources (e.g., multiple I/O buffers or multi-page memory), our algorithm is more suitable for real-world systems.

III. Background with Definitions

In this section, we introduce some concepts from graph theory and operating system, which play important roles in our algorithm. Above all, we use a Resource Allocation Graph (RAG) to represent the resource allocation scenario among processes and resources in a system. We first describe some definitions related to RAG.

Definition 3.1: A RAG is a *bipartite* directed graph that consists of a set of process nodes, a set of resource nodes and a set of edges each of which is either directed from a process node to a resource node (a request edge) or from a resource node to a process node (a grant edge).
The set of nodes and the set of edges are denoted by V and E, respectively.

Definition 3.2: A path (p_{i_1}, q_{j_1}), (q_{j_1}, p_{i_2}), ..., (p_{i_k}, q_{j_k}), ..., $(q_{i_s}, p_{j_{s+1}})$, is a sequence of alternating nodes connected by edges where each edge is distinct, and where p_{i_k} denotes a process node and q_{j_k} denotes a resource node. The length of a path is the number of edges along the path.

If any nodes in a path are not distinct, it contains at least one cycle. A cycle does not contain any sink nodes (see Definition 3.7 for the formal definition of a sink node).

Definition 3.3: A simple path is a path in which all nodes along the path are distinct. Thus, a simple path does not contain any cycle.

Definition 3.4: The distance from v_i to v_j is the number of edges along a simple path that starts from v_i and ends at v_j.

While the aforementioned RAG representation is suitable for single-unit resource systems, it is inadequate for multi-unit resource systems. In a multi-unit resource system, a process may request and thus be assigned not only multiple resources but also more than one unit of the same type of resource. To carry the information on how many units of a resource are assigned to a specific process, the RAG is modified such that every grant edge $e_{ji} = (q_j, p_i)$ is given a number that is equal to the number of units of resource q_j assigned to process p_i. We call this RAG, which carries the numbers of assigned units, a *weighted RAG*.

Then, let us introduce four key concepts used in our algorithm.

Definition 3.5: A node v_i is reachable from a node v_j if and only if there exists a path that starts from v_j and ends at v_i. Thus, v_i is called a reachable node of v_j [5].

Definition 3.6: The reachable set of node v is a set of nodes such that a path exists from v to every node in the reachable set [5].

The reachable set of node v is denoted as $\gamma(v)$.

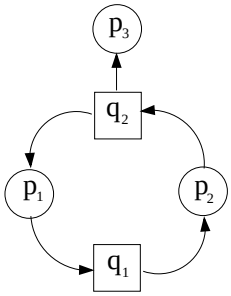
Definition 3.7: A sink node is a node that does not have any outgoing edge [3], [5].

Definition 3.8: A knot is a nonempty set K of nodes such that the reachable set of each node in K is exactly K [5].

The following example illustrates the concept of reachable sets and knots.

Example 1: Examples of RAGs with and without knots.

Consider the resource allocation situation in a system represented by Figure 1(a). The reachable set for each and every node in Figure 1(a) is shown in Figure 1(b):



(a) A RAG without a knot.

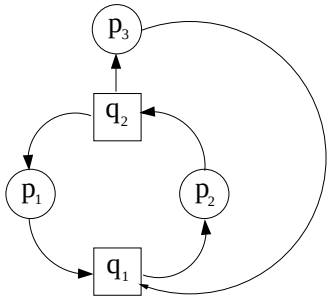
$\gamma(p_1)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(p_2)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(p_3)=\{\}$
$\gamma(q_1)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(q_2)=\{p_1, q_1, p_2, q_2, p_3\}$

(b) The reachable set for each node.

Figure 1. An example of a RAG without any knot.

As can be seen, there exists no set of nodes that fulfills the definition of a knot. Note that p_3 in Figure 1(a)

is a sink node because it does not have any outgoing edge. Next, let us insert an edge (p_3, q_1) as shown in Figure 2(a). The reachable set for each and every node in Figure 2(a) is shown in Figure 2(b).



(a) A RAG with a knot.

$\gamma(p_1)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(p_2)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(p_3)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(q_1)=\{p_1, q_1, p_2, q_2, p_3\}$
$\gamma(q_2)=\{p_1, q_1, p_2, q_2, p_3\}$

(b) The reachable set for each node.

Figure 2. An example of a RAG with a knot.

Now in Figure 2(b), let us consider a set K containing all the nodes in the RAG (i.e., K=$\{p_1, q_1, p_2, q_2, p_3\}$). The reachable set for each and every node in K is exactly K itself. Thus, set K is a knot. ■

Next, a *single-unit resource*, a *multi-unit resource* and an *expedient* state are defined as follows:

Definition 3.9: A single-unit resource is a resource that serves at most one process at a time. This means while a single-unit resource is serving a process, all other processes requesting this resource must wait [20].

Definition 3.10: A multi-unit resource is a resource that can serve one or more processes at the same time. All processes are served with the same or similar functionality [20].

Note that the multi-unit resource is a generalized concept of the single-unit resource. A multi-unit resource continues to serve new requests as long as it has enough free units to be assigned.

Definition 3.11: A system is in an *expedient* state if all requests for available units are granted immediately [5].

Thus, if a process requests an available unit of a resource, thereby forming a request edge, the request edge is immediately replaced by a grant edge after the unit is granted. That is, we do not consider the intermediate state with a temporary request as a meaningful state.

To facilitate the computations in the algorithm, we also define a *sink bitmask* and a *process bitmask* as follows.

Definition 3.12: A sink bitmask for the reachable sink nodes of resource q_j is an m-bit vector $[0...0101...0]^T$ in which its i^{th} digit is 1 if process p_i is a reachable sink node from q_j in the RAG, where $1 \leq i \leq m$. Otherwise, the i^{th} digit is 0.

Definition 3.13: The process bitmask of process p_i is an m-bit vector $[0...010...0]^T$ in which only the i^{th} bit is 1, and all others are zeros.

IV. The Theory Behind Deadlock Detection in Multi-unit Resource Systems

In this section, we introduce two necessary and sufficient conditions for deadlock with respect to a RAG. It is well known that a cycle in a RAG is a necessary and sufficient condition for deadlock for single-request systems consisting of only single-unit resources [5], [10], [11], [18]. However, in multi-unit resource systems, a cycle in a RAG is no longer a sufficient condition for deadlock. The following theorem introduces a necessary and sufficient condition for deadlock in a multi-unit resource system.

Theorem 4.1: An expedient multi-unit resource system with single-unit requests is in a deadlock state if and only if its corresponding RAG contains a knot [5].

By Theorem 4.1, detecting deadlock in a multi-unit resource system is equivalent to finding a knot in its corresponding RAG. It is observed that if a resource is requested by a process whose corresponding node is the only reachable sink node for that resource node, a knot forms in the RAG. Based on this observation, Kim proposed an algorithm that is capable of detecting deadlock in $\mathcal{O}(1)$ run-time [9]. We formalize such an observation by the following theorem.

Theorem 4.2: In an expedient system that is not in a deadlock state, when process p_i requests a unit of resource q_j whose unit is unavailable, its request is blocked. Then, the edge (p_i, q_j) is inserted in its RAG. In the new RAG, the reachable set of q_j is a knot if and only if p_i was the only sink node in the reachable set of q_j before p_i requests q_j.

To make the proof of Theorem 4.2 more concise, we first introduce and prove Lemma 4.3.

Lemma 4.3: If there exists a path from node v_i to node v_j, then $\gamma(v_j) \subseteq \gamma(v_i)$.

Proof: Let us assume that node v_k is any of v_j's reachable nodes, i.e., there exists a path from v_j to v_k. Then, because a path exists from v_i to v_j, there must exist a path from v_i to v_k ($v_i \rightarrow \ldots \rightarrow v_j \rightarrow \ldots \rightarrow v_k$). Thus, v_k is also a reachable node of v_i. That is, any reachable node of v_j is also a reachable node of v_i. Since v_i may have other reachable nodes that are located in between v_i and v_j, the reachable set of v_j is a subset of the reachable set of v_i. It is possible that there exist other paths directed from v_i to more nodes, which are not reachable by v_j. In such cases, the reachable set of v_j is a subset of the reachable set of v_i as well. ■

To prove Theorem 4.2, we also utilize Lemma 4.4 from [5].

Lemma 4.4: A graph does not contain knots if and only if for each node (say v_i) in the graph, either v_i itself is a sink or there exists a path directed from v_i to a sink node.

Proof: See [5] for the proof of Lemma 4.4 ■

Now, let us prove Theorem 4.2.

Proof:

1) $\Rightarrow$(the sufficient condition): Suppose that edge (p_i, q_j) is inserted in the RAG and that makes the reachable set of q_j a knot. We will show that p_i is the only reachable sink node of q_j before p_i requests q_j by contradiction. Let us assume that p_i is not the only reachable sink node of q_j; i.e., there exists another sink node (say p_i') reachable from q_j. Since the reachable set of q_j becomes a knot after p_i requests q_j, and since p_i' is a reachable node of q_j, the reachable set of p_i' is also the reachable set of q_j by the definition of knot (Definition 3.8). However, p_i' is a sink node, which means that the reachable set of p_i' is an empty set according to Definition 3.5 and 3.7. This contradicts the conclusion that the reachable set of p_i' is the same as the reachable set of q_j, which is non-empty. This contradiction results from the assumption that p_i is not the only reachable sink node of q_j. Therefore, p_i is the only reachable sink node of q_j before p_i requests q_j.
2) $\Leftarrow$(the necessary condition): Suppose that p_i is the only reachable sink node of q_j before p_i requests q_j. We will show that the reachable set of q_j is a knot after edge (p_i, q_j) is inserted in the RAG. Let us assume, before p_i requests q_j, n_k is any of q_j's reachable nodes. Thus, according to Lemma 4.3, $\gamma(n_k) \subseteq \gamma(q_j)$. Because the RAG does not contain any knot, by Lemma 4.4, n_k is either a sink node or has a path directed from n_k to a sink node. Then, n_k either (i) is p_i or (ii) has a path directed to p_i. It is because if n_k satisfies neither (i) nor (ii), q_j has more than one reachable sink nodes. Hence, after edge (p_i, q_j) is inserted in the RAG, there exists a path from n_k to q_j as well ($n_k \rightarrow \ldots \rightarrow p_i \rightarrow \ldots \rightarrow q_j$). According to Lemma 4.3, $\gamma(q_j) \subseteq \gamma(n_k)$. Hence, $\gamma(n_k) = \gamma(q_j)$. That is, for any reachable node n_k of q_j, the reachable set of n_k is the reachable set of q_j. By Definition 3.8, the reachable set of q_j is a knot. ■

Our $\mathcal{O}(1)$ deadlock detection requires that all reachable sink nodes for all resources have been pre-computed and known at the time a resource request is blocked. We call this pre-computation of the reachable sink nodes for all resources *detection preparation*. Detection preparation in Kim's algorithm requires a run-time complexity of $\mathcal{O}(m \times n)$. Unlike his algorithm, our novel detection preparation method conducts detection preparation for all resources by performing Boolean arithmetic among the elements

of several matrices, and thus parallelizes the entire preparation with respect to each resource. As a result, the run-time complexity of detection preparation is reduced to $\mathcal{O}(min(m, n))$. The following example illustrate the gist of our algorithm by showing how each part of a RAG is traversed for each resource to find its reachable sink nodes.

Example 2: Traversing a RAG to search for sink nodes.

Let us reconsider Figure 1(a) as an example. There are two resources in the RAG, namely q_1 and q_2. Figure 3(a) and (b) show the steps that are taken to search for the reachable sink nodes of q_1 and q_2, respectively. Now, let us focus on the q_1's search. The left-hand side of Figure 3(a) shows the first iteration of the q_1's search in the form of graph traversing. Only the edges that are traversed in the first iteration are shown in the graph; others are omitted. The process and resource nodes visited in the first iteration are highlighted (i.e., process node p_2 and resource node q_2). From Figure 3(a), we can see that no reachable sink node of q_1 is found in the first iteration. Similarly, the second iteration of the search is depicted on the right-hand side of Figure 3(a). It shows that in the second iteration the search starts from q_2 (i.e., the point where the first iteration stopped) and visits process nodes p_1, p_3 and resource node q_1. Therefore, sink node p_3 is found to be reachable from q_1. Now, the search for q_1's reachable sink nodes either cannot travel farther because of encountering a sink node (i.e, p_3), or arrives at a node previously visited (i.e., q_1). This indicates that the search for q_1's reachable sink nodes is complete. In the same way, the two iterations of the search for q_2's reachable sink nodes are illustrated in Figure 3(b). From the search, we find p_3 to be the only reachable sink node for q_2 as well. The searches for q_1 and q_2's reachable sink nodes can be performed in parallel because there is no dependency between them. After the searches are complete, the information on the reachable sink nodes for each resource is stored and ready for next deadlock detection.

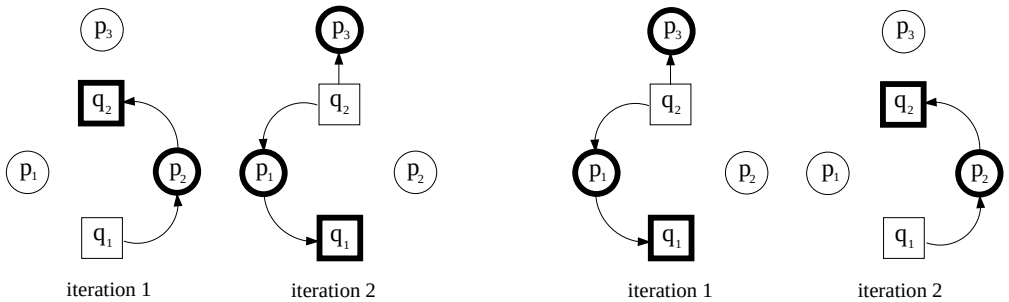


Figure 3. Examples of the steps in search for sink nodes.

V. A New Deadlock Detection Method for Multi-unit Resource Systems

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

A. Assumptions

For the presented deadlock detection algorithm, we make the following assumptions for its applicable systems:

1) Each type of resource has a fixed total number of units.
2) The system is in an *expedient* state (see Definition 3.11 for the formal definition of an expedient state).
3) A process requests one resource unit at a time. Thus, a process is blocked as soon as it requests an unavailable resource. By Theorem 4.1, a knot becomes a necessary and sufficient condition for deadlock [5].

B. Introduction of a Matrix Representation of a Weighted RAG

Commonly, a graph is represented by an adjacency matrix [4]. Specifically, a RAG can be represented by an $(m+n) \times (m+n)$ matrix, where m is the number of processes and n is the number of resources. Each element of the matrix is either 1 or 0, with 1 indicating an edge directed from the node labeled by the row to the node labeled by the column, and 0 indicating no edge. Such an adjacency matrix for a RAG is defined as follows: $A = \{A[i][j]\}_{(m+n)\times(m+n)}$, $(1 \leq i \leq (m+n), 1 \leq j \leq (m+n))$, where

$$A[i][j] = \begin{cases} 1 & \text{if } \exists(node_i, node_j) \in \text{E}, \\ 0 & \text{otherwise.} \end{cases}$$

Due to the bipartite property of a RAG, if we arrange the nodes such that process and resource nodes appear consecutively among rows and columns in the matrix, we obtain two zero sub-matrices and the other two non-zero ones, as shown in Figure 4. Therefore, it is sufficient to use only the two non-zero sub-matrices to represent a complete RAG, one (denoted as *AG* in Figure 4) representing grant edges and the other (denoted as *AR*) representing request edges.

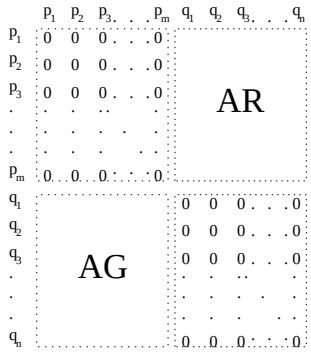


Figure 4. A RAG Adjacency Matrix.

In summary, sub-adjacency matrices for a RAG(V, E) can be defined as follows: $AG = \{AG[j][i]\}_{n \times m}$, $(1 \leq j \leq n, 1 \leq i \leq m)$, $AR = \{AR[i][j]\}_{m \times n}$, $(1 \leq i \leq m, 1 \leq j \leq n)$, where m is the number of processes and n is the number of resources, and where

$$AG[j][i] = \begin{cases} 1 & \text{if } \exists(q_j, p_i) \in \text{E}, \\ 0 & \text{otherwise}, \end{cases}$$

$$AR[i][j] = \begin{cases} 1 & \text{if } \exists(p_i, q_j) \in \text{E}, \\ 0 & \text{otherwise}. \end{cases}$$

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 sub-adjacency matrix AR can contain at most one non-zero 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 all zero values. Such a process node is a sink in the RAG (see Definition 3.7). A sink node represents an active process because the process has acquired all the necessary resources to run.

As stated in Section III, we use a weighted RAG to represent resource allocation in a multi-unit resource system, where each grant edge has a weight (i.e., the number of assigned units). To properly represent such a weighted RAG, in addition to the sub-adjacency matrices, we introduce another n by m 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.

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

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

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

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

The following example shows a weighted RAG and the matrices representing it.

Example 3: A weighted RAG, its sub-adjacency matrices, weight matrix and free-unit vector.

A weighted RAG for a 3x3 system with multi-unit resources is shown in Figure 5. The matrices that we use to represent such a weighted RAG are shown in Figure 6. We assume q_1 has 3 units in total; q_2 has 2 units in total; q_3 has 4 units in total. AG presents all grant edges in the weighted RAG. For example, $AG[q_1][p_1]$ is 1 because some units of q_1 are granted to p_1. The exact number of units of q_1 granted to p_1 is present in $W[q_1][p_1]$ (i.e., 2). AR presents all request edges in the weighted RAG. For example, $AR[p_2][q_2]$ is 1 because p_2 is blocked on its request for a unit of q_2. Also note that the sum of all elements in one row of W plus its corresponding element in F is equal to the total number of resource units. For instance, all elements in the 3^{rd} row (for q_3) in W and the 3^{rd} element in F add up to 4, which is the total number of units of q_3. If an element in F is 0, it indicates that there remains no free unit of such resource currently. For instance, q_2 does not have any free units, thus $F[2]$ is 0. ■

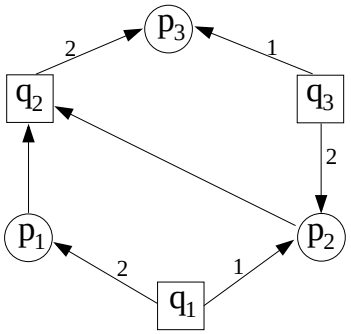


Figure 5. A weighted RAG with 3 processes and 3 multi-unit resources.

C. Our Algorithm and Its Detail

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. These

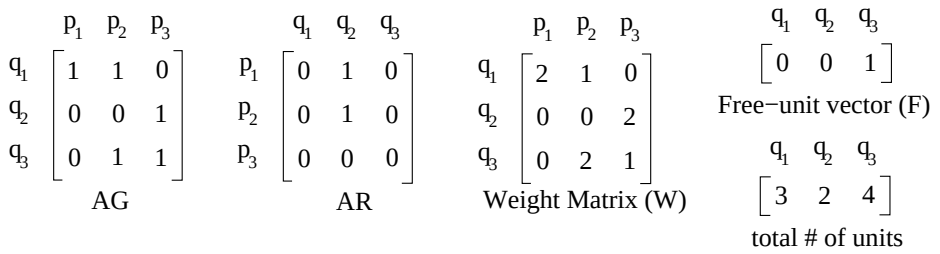


Figure 6. Examples of matrices AG, AR, Weight Matrix and Free-unit Vector

Table 1. Data structures for MDDA.

name	notation	description
AG[j][i] in $AG_{n\times m}$	ag_{ji}	whether any units of resource j are granted to process i
AR[i][j] in $AR_{m\times n}$	ar_{ij}	whether any request for resource j from process i is blocked
Weights[j][i] in $W_{n\times m}$	w_{ji}	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 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][l] in $WR_{n\times n}$	wr_{jl}	whether resource l is being visited in the current step during the search for resource j's $sink$ nodes
PrevResource[j][l] in $PR_{n\times n}$	pr_{jl}	whether resource l has been visited in any previous steps during the search for resource j's $sink$ nodes

data structures include seven matrices and two vectors. Among the data structures, a set of matrices AG[], AR[] and Weights[] as well as a set of vectors FreeUnit[] and SinkProcess[] are used to store information representing a weighted RAG. A set of matrices Sink[], WorkProcess[], WorkResource[] and PrevResource[] are used to store information on search results during the intermediate steps of MDDA. In this report, 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.

Two sub-adjacency matrices (AG[] and AR[]) for a RAG, a weight matrix (Weights[]) and a free-unit vector (FreeUnit[]) have been defined in Section V-B. 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:

$SP = [sp_i]_m$, $(1 \leq i \leq m)$,

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

The matrices mentioned so far always maintain their states whereas the following four matrices are initialized every time *Step 4* of Algorithm 5.3 is entered.

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

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

WorkProcess[] and WorkResource[] are used to store information about the process and resource nodes visited during the intermediate steps of the search for sink nodes, 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: $WR = [wr_{jl}]_{n \times n}$, $(1 \leq j \leq n, 1 \leq l \leq n)$,

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

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: $PR = [pr_{jl}]_{n \times n}$, $(1 \leq j \leq n, 1 \leq l \leq n)$,

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

Note that a resource can have multiple sink nodes. Thus each column of the sink-ID matrix represents all the reachable sink nodes of a resource. We call this type of binary representation *sink bitmask* or *sink*

node bitmask.

Definition 5.1: A sink bitmask for the reachable sink nodes of resource q_j is an m-bit vector $[0...0101...0]^T$ in which its i^{th} digit is 1 if process p_i is a reachable sink node from q_j in the RAG, where $1 \leq i \leq m$. Otherwise, the i^{th} digit is 0.

To facilitate the computations in our algorithm, we also define here the bitmask of a process.

Definition 5.2: The bitmask of process p_i is an m-bit vector $[0...010...0]^T$ in which only the i^{th} digit is 1, and all others are zeros.

Note that the sink bitmask for the reachable sink nodes of a resource and the bitmask of a process are comparable and eligible for any bit-wise operations since both are m-digit binary vectors. Algorithm 5.3 is the pseudo code of our algorithm, followed by explanation.

Algorithm 5.3: Multi-unit resource Deadlock Detection Algorithm (MDDA)

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 *AG[j][i] = 1* // no deadlock results
5 *increase Weights[j][i] by one*
6 *decrease FreeUnit[j] by one*
7 *go to Step 4*
8 *else*
9 *AR[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 AG[], 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 sink bitmask for q_j's sink nodes)*
// p_i is the only $sink$ node reachable from q_j

16 *deadlock exists*

17 *else*

18 *no deadlock exists*

19 *go to Step 4*

Step 4: Update sink bitmasks

20 *for all* $i = 1, \ldots, m,\ SinkProcess[i] = \sim(\vee_{1 \le t \le n} AR[i][t])$

21 *for all* q_j *(j=1,…,n)* // do in parallel

22 $Sink[][j]^{(0)} = [0]_{m \times 1}$; $WorkProcess[][j]^{(0)} = [0]_{m \times 1}$

23 *if AG[j][] contains at least one non-zero element*

24 *for all* $l = 1, \ldots, n, WorkResource[j][l]^{(0)} = 1$ *if* $l == j$; 0 *otherwise*

25 *for all* $l = 1, \ldots, n, PrevResource[j][l]^{(0)} = 1$ *if* $l == j$; 0 *otherwise*

26 *for (k=1; k≤min(m,n); k++)* // track q_j's sink id's

27 *for all* $i = 1, \ldots, m, WorkProcess[i][j]^{(k)} = \vee_{1 \le t \le n}(WorkResource[j][t]^{(k-1)} \wedge AG[t][i])$

28 *for all* $i = 1, \ldots, m, Sink[i][j]^{(k)} = (WorkProcess[i][j]^{(k)} \wedge SinkProcess[i]) \vee Sink[i][j]^{(k-1)}$

29 *for all* $l = 1, \ldots, n, WorkResource[j][l]^{(k)}$
$= \vee_{1 \le c \le m}(WorkProcess[c][j]^{(k)} \wedge AR[c][l]) \wedge (\sim PrevResource[j][l]^{(k-1)})$

30 *for all* $l = 1, \ldots, n, PrevResource[j][l]^{(k)} = WorkResource[j][l]^{(k)} \vee PrevResource[j][l]^{(k-1)}$

31 *if* $WorkResource[j][]^{(k)}$ *contains all zeros*

32 *stop for* q_j

* Lines 27 to 30 are computed in sequence. $Element[i][j]^{(k)}$ denotes its value right after the computation in the k^{th} iteration is finished; $Element[i][j]^{(0)}$ denotes its initial value.

* $\sim$, $\vee$ and $\wedge$ denote NOT, bit-wise OR and AND, respectively.

In *Step 1*, MDDA handles the situation where process p_i requests a unit of resource q_j. MDDA first checks FreeUnit[j]. If FreeUnit[j] is greater than zero, meaning resource q_j has available units, a unit of q_j is assigned to p_i, and all related matrix elements are updated. Since there should be no deadlock caused by resource grants, MDDA skips deadlock detection in *Step 3*, and updates the sink bitmasks for all resources in *Step 4*. The computations performed in *Steps 3* and *4* are elaborated in Section V-D. On the other hand, if FreeUnit[j] is zero, which signifies that all units of resource q_j are being used, MDDA enters *Step 3* to check for potential deadlock. If MDDA does not find deadlock, it updates the sink bitmasks for all resources in *Step 4*. In the event a unit of resource q_j is released from process p_i, MDDA enters *Step 2*. If process p_s is waiting for the released unit of resource q_j, MDDA enters *Step 1* (i.e., Resource request event (p_s, q_j)), and the same procedures afore-stated occur. If that resource unit

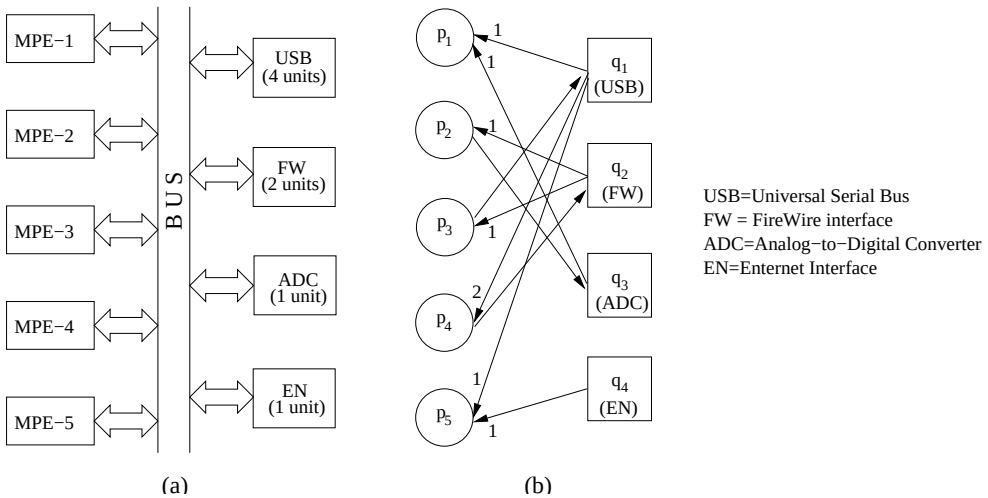


Figure 7. SoC Example

becomes free, it is possible that the RAG is changed. Thus, MDDA enters *Step 4* to update the sink bitmasks for all resources.

D. MDDA in a Particular Scenario

In this section, we illustrate our algorithm, especially *Step 4*, with an example system consisting of five processors and four multi-unit resources, shown in Figure 7. We assume that one process is running on each processor. The example is also used for one of our experimental simulations. Figure 7(b) shows a current resource allocation status in the form of a weighted RAG. Note that the number of total units for each resource is specified in Figure 7(a).

We will show how the sink node bitmasks are computed for each resource in *Step 4*. The resulting sink node bitmasks are saved in Sink[] and then used for future deadlock detection in *Step 3*. Our algorithm stores the sink bitmask for reachable sink nodes of resource q_j in Sink[][j] (i.e., j^{th} column of Sink[]).

The following matrices AG[], AR[], Weights[], FreeUnit[] and SinkProcess[] together show the current resource allocation status in the system (abbreviated notations in Table 1 are used):

$$\mathbf{AG_{n\times m}} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{AR_{m\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{W_{n\times m}} = \begin{bmatrix} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{F_n} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{SP_m} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \end{bmatrix}^T$$

$AG_{n\times m}$ stores the grant edges from resources to processes shown in Figure 7(b). $AR_{m\times n}$ stores the request edges from processes to resources shown in Figure 7(b). $W_{n\times m}$ stores the numbers of units of each resource assigned to each process. F_n indicates that all units of every resource have been granted; thus, there exists no available resource units in the current system. SP_m signifies that in the current RAG, processes p_1 and p_5 are sink nodes. Note that SP_m is computed from $AR_{m\times n}$ using the following equation:

$$\forall i \; sp_i = \sim(\vee_{1\leq t\leq n} ar_{it}), \tag{1}$$

where $\sim$ and $\vee$ denote NOT and bit-wise OR, respectively. For instance, to determine whether or not p_5 is a sink, we calculate $SP[5]$ as follows: $SP[5]$ = $\sim$ $(AR[5][1] \vee AR[5][2] \vee AR[5][3] \vee AR[5][4]$ = $\sim (0 \vee 0 \vee 0 \vee 0)$ = 1. From the value of $SP[5]$, we know that p_5 is a sink node.

Those resources that have all units available have no edges connected in the RAG. Thus, their corresponding columns in Sink[] are merely assigned zeros, and they are excluded from the computation of sink bitmasks. Hence, the computations of sink node bitmasks are performed for the rest of resources, each of which has at least one grant edge (i.e., outgoing edge).

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 zero (SK, WP = $[0]_{m\times n}$). WorkResource[] and PrevResource[] are initialized as follows:

$$\mathbf{WR^{(0)}_{n\times n}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{PR^{(0)}_{n\times n}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 the 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 AG[] (line 27). The operation in line 27 is equivalent to the following equation:

$$\forall i, j \; wp^{(k)}_{ij} = \vee_{1\leq t\leq n} (wr^{(k-1)}_{jt} \wedge ag_{ti}), \tag{2}$$

where $\vee$ and $\wedge$ denote bit-wise OR and AND, respectively. 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., the j^{th} row) and storing them in WorkProcess[][j].

Now let us choose resource 1 (q_1) of Figure 7 as an example to discover its $sink$ nodes and take a closer look at how the matrices are computed. In the RAG of Figure 7(b), the units of q_1 are assigned to three processes (p_1, p_4 and p_5). At the same time, p_3's request for q_1 is blocked. Accordingly, AG[1][] is $[10011]$, and AR[][1] is $[00100]^T$. Before the first iteration, WorkResource[1][] was initialized as $[1000]$ (line 24). Thus, WorkProcess[][1] is first computed and stored as bit pattern $[10011]^T$ (line 27) since p_1, p_4 and p_5 are directly connected from q_1. For the purpose of illustration, the calculation of $WP[4][1]^{(1)}$ is shown here: $WP[4][1]^{(1)} = (WR[1][1]^{(0)} \wedge AG[1][4]) \vee (WR[1][2]^{(0)} \wedge AG[2][4]) \vee (WR[1][3]^{(0)} \wedge AG[3][4]) \vee (WR[1][4]^{(0)} \wedge AG[4][4]) = (1 \wedge 1) \vee (0 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 0) = 1$.

In the same way, for the rest of resources, their corresponding columns in WorkProcess[] (WorkProcess[][2] through WorkProcess[][4]) are computed. The following values of WorkProcess[] result:

$$\mathbf{WP^{(1)}_{m \times n}} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

WorkProcess[][2] is computed as $[01100]^T$ because p_2 and p_3 are directly connected from q_2, as can be seen in Figure 7(b). Similarly, WorkProcess[][3] is $[10000]^T$ since only p_1 is directly connected from q_3. WorkProcess[][4] is $[00001]^T$ because only p_5 is directly connected from q_4.

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:

$$\forall i, j \; sk_{ij}^{(k)} = (wp_{ij}^{(k)} \wedge sp_i) \vee sk_{ij}^{(k-1)}. \tag{3}$$

For the case of q_1, the algorithm checks whether anyone is a $sink$ node among the three process nodes (p_1, p_4 and p_5) by comparing WorkProcess[][1] ($[10011]^T$) with SinkProcess ($[10001]^T$) (line 28). For instance, $SK[][1]^{(1)}$ is calculated as follows: $SK[][1]^{(1)} = (WP[][1]^{(1)} \wedge SP[]) \vee SK[][1]^{(0)} = ([10011]^T \wedge [10001]^T) \vee [00000]^T = [10001]^T$.

As a result, p_1 and p_5 are identified as newly found sink nodes for resource q_1. Thus, they are saved in Sink[][1] (1^{st} and 5^{th} elements of the 1^{st} column become ones). In the same way, the rest

of Sink[] (Sink[][2] through Sink[][4]) are computed for resource 2 through resource 4. The following values of Sink[] result:

$$\mathbf{SK}^{(1)}_{\mathbf{m\times n}} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Sink[][2] ($[00000]^T$) indicates that so far the algorithm has found no sink nodes reachable from q_2. By Sink[][3] ($[10000]^T$) and Sink[][4] ($[00001]$), we know that sink nodes p_1 and p_5 are reachable from q_3 and q_4, respectively.

Next, the algorithm computes WorkResource[] from WorkProcess[] (just computed), AR[] 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 q_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:

$$\forall j, l\ wr^{(k)}_{jl} = (\vee_{1\leq c\leq m}(wp^{(k)}_{cj} \wedge ar_{cl})) \wedge (\sim pr^{(k-1)}_{jl}). \tag{4}$$

To illustrate the computation of WorkResource[], let us continue with the example for q_1. WorkProcess[][1] ($[10011]^T$) indicates that q_1 now reaches three process nodes (p_1, p_4 and p_5). From these three process nodes, p_4 has one outgoing edge (a request edge) to q_2; p_1 and p_5 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_4 since p_4 is blocked for q_2. For instance, $WR[1][2]^{(1)}$ can be calculated as follows: $WR[1][2]^{(1)} = ((WP[1][1]^{(1)} \wedge AR[1][2]) \vee (WP[2][1]^{(1)} \wedge AR[2][2]) \vee (WP[3][1]^{(1)} \wedge AR[3][2]) \vee (WP[4][1]^{(1)} \wedge AR[4][2]) \vee (WP[5][1]^{(1)} \wedge AR[5][2])) \wedge (\sim PR[1][2]^{(0)}) = ((1 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 0) \vee (1 \wedge 1) \vee (1 \wedge 0)) \wedge 1 = 1$.

Likewise, the rest of WorkResource[] (WorkResource[2][] through WorkResource[4][]) are computed for resource 2 through resource 4. The following values of WorkResource[] result:

$$\mathbf{WR}^{(1)}_{\mathbf{n\times n}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This WR indicates the following as can be seen from Figure 7(b): q_1 can reach q_2 via p_4; q_2 can reach q_1 and q_3 via p_3 and p_2, 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:

$$\forall j, l \; pr_{jl}^{(k)} = wr_{jl}^{(k)} \vee pr_{jl}^{(k-1)}. \tag{5}$$

Continue with the example of q_1. PrevResource[1][] was initialized as $[1000]$ (line 25). Currently, the values of WorkResource[1][] are $[0100]$, meaning the search for q_1's $sink$ nodes is now visiting q_2 in the current iteration. Thus, the new values of PrevResource[1][] is computed as $[1100]$ (line 30), which signifies both q_1 and q_2 have now been visited (1 indicates "visited"). For example, $PR[1][2]^{(1)}$ is calculated as follows: $PR[1][2]^{(1)} = WR[1][2]^{(1)} \vee PR[1][2]^{(0)} = 1 \vee 0 = 1$. This mechanism effectively prevents our algorithm from being trapped in *cycles*.

The rest of PrevResource[] (PrevResource[2][] through PrevResource[4][]) are computed for resource 2 through resource 4 in the same way. The following values of PrevResource[] result:

$$\mathbf{PR}_{\mathbf{n\times n}}^{(\mathbf{1})} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix indicates that starting from q_1, the search has visited q_2 in addition to q_1 itself; from q_2, the search has visited q_1 as well as q_3 in addition to q_2 itself.

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). Note that WorkResource[j][l] is computed using the values of WorkProcess[][j] in the current iteration and AR[][l] as well as PrevResource[j][l] in the last iteration. Thus, in the search for the reachable sink nodes of resource q_j, WorkResource[j][l] being zero means either of the following: i) there are no edges directed from the process nodes computed in the current iteration (i.e., WorkProcess[][j]) to q_l; ii) resource q_l has been visited in a previous iteration. If resource q_l satisfies neither condition (i) nor (ii), WorkResource[j][l] is 1. That is, there exist more nodes to be visited, and some sink nodes reachable from q_j (via q_l) may remain unvisited, and thus the search for

q_j's sink nodes continues. On the other hand, if WorkResource[j][] contains all zeros, the search for q_j's sink nodes is finished. As a result, Sink[][j] contains a complete sink bitmask for q_j's reachable sink nodes. 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 bitmasks. This informs that q_3 and q_4 have their reachable sink nodes as p_1 and p_5, 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_4 and p_5 (which are stored as bitmask $[10011]^T$ 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_2 and p_3 ($[01100]^T$ 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:

$$\mathbf{WP^{(2)}_{m\times n}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{SK^{(2)}_{m\times n}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \quad \mathbf{WR^{(2)}_{n\times n}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{PR^{(2)}_{n\times n}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now for q_2, the 2^{nd} row of WR also contains all zeros, which means the search for q_2's sink nodes has visited all nodes in q_2's reachable set. The resource nodes visited so far for q_2 are signified by the 2^{nd} row in PR (i.e., q_1, q_2 and q_3; 1 means 'visited'). Then, q_2's column in SK (the 2^{nd} column) contains its complete sink bitmask, which means that q_2's sink processes are p_1 and p_5. Furthermore, the corresponding column in WP for q_2 (i.e., the 2^{nd} column) indicates that p_1, p_4 and p_5 are the latest process nodes visited during the search for q_2's sink nodes. Note that the 3^{rd} and 4^{th} columns in SK, as well as the 3^{rd} and 4^{th} rows in WR and PR, are not updated because q_3 and q_4 have found all their sink process nodes at the last iteration. Accordingly, the 3^{rd} and 4^{th} columns in WP contains all zeros, which signify that the searches for p_3 and p_4's sink nodes have been stopped. Note that one of the paths starting from

q_2 becomes a cycle in the second iteration (i.e., $(q_2, p_3, q_1, p_4, q_2)$). However, re-visiting q_2 is prohibited by involving matrix PR in the computations of matrix WR (i.e., Equation 4). $WR[2][2]$ is computed as follows: $WR[2][2]^{(2)} = ((WP[1][2]^{(2)} \wedge AR[1][2]^{(2)}) \vee (WP[2][2]^{(2)} \wedge AR[2][2]^{(2)}) \vee (WP[3][2]^{(2)} \wedge AR[3][2]^{(2)}) \vee (WP[4][2]^{(2)} \wedge AR[4][2]^{(2)}) \vee (WP[5][2]^{(2)} \wedge AR[5][2]^{(2)})) \wedge (\sim PR[2][2]^{(1)}) = ((1 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 0) \vee (1 \wedge 1) \vee (1 \wedge 0)) \wedge 0 = 0$. Note that without $PR[2][2]^{(1)}$, $WR[2][2]^{(2)}$ would become 1, which makes the search for the sink nodes of q_2 start over from q_2 and repeat all the computations performed in the 1^{st} and 2^{nd} iterations. From the computation of $WR[2][2]^{(2)}$, we can see that the purpose of the use of matrix PR in Equations 4 and 5 is to guarantee that the search for each resource's sink nodes only visits each resource node in its reachable set once, which ensures that the run-time complexity of updating all sink bitmasks (i.e. detection preparation) is bounded by $\mathcal{O}(min(m, n))$.

For q_1, however, its corresponding row of matrix WR (the 1^{st} row) still contains 1 at its 3^{rd} element. This means that the search for q_1's sink nodes now needs to step further from q_3. Thus, further iterations must be performed to track more sink nodes. After another iteration of computation is performed for q_1, the following results:

$$\mathbf{WP}^{(3)}_{\mathbf{m\times n}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{SK}^{(3)}_{\mathbf{m\times n}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \quad \mathbf{WR}^{(3)}_{\mathbf{n\times n}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{PR}^{(3)}_{\mathbf{n\times n}} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now for q_1, its row (1^{st} row) of WR also contains all zeros. It means that its column in SK (1^{st} 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_1 and p_5, while q_3 and q_4 are bonded to their single sink nodes p_1 and p_5, respectively, as also can be seen in Figure 7(b). The resulting sink process nodes for each resource is listed in Table 2.

Table 2. The Reachable Sink Nodes for SoC Example

	q_1(USB)	q_2(FW)	q_3(ADC)	q_4(EN)
Sink node	p_1, p_5	p_1, p_5	p_1	p_5

Through the explanation, we have seen how MDDA updates the information on reachable sink nodes for all resources. That makes MDDA ready for a new detection invocation for the next resource event. Let us now continue from the current state of Figure 7(b) and consider that p_1 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_5, this request is blocked. Thus, p_1 must wait for q_4 to become free. Then, the deadlock detection unit is invoked immediately (*Step 3*). As indicated in our algorithm (line 15), we simply compare q_4's sink bitmask Sink[][4] ($[00001]^T$) with the bitmask of p_1 ($[10000]^T$). Since the two bitmasks are different, we can conclude that this request does not cause deadlock in the system. Accordingly, the sink bitmasks for all resource nodes are updated in *Step 4*. The new reachable sink nodes for each resource are listed in Table 3.

Table 3. The New Reachable Sink Nodes for SoC Example

	q_1(USB)	q_2(FW)	q_3(ADC)	q_4(EN)
Sink node	p_5	p_5	p_5	p_5

Let us further consider that p_5 makes a request for a unit of q_2 after p_1 is blocked for q_4 as afore-stated. Does the new request cause deadlock? p_5 is blocked for this request because q_2 has two units being used by p_2 and p_3. The sink bitmask of q_2 ($[00001]^T$) contains only one 1 at the 5^{th} element, which signifies that before p_5 makes a request for a unit of q_2, p_5 is the only reachable sink node from q_2. This fulfills the condition stated in Theorem 4.2 for finding a knot in the RAG. Then, by Theorem 4.1, we can conclude the system enters a deadlock state. This is immediately determined in *Step 3* of Algorithm 5.3 by comparing the sink bitmask for q_2 ($[00001]^T$) with the bitmask of process p_5 ($[00001]^T$).

VI. Proof of the Correctness and Run-time Complexity of MDDA

So far, we have described MDDA in detail and explained its deadlock detection process with an example. In this section, we show the proofs of the correctness and run-time complexity of MDDA.

A. The Problem Statement

MDDA consists of four steps as shown in Algorithm 5.3. Steps 1 and 2 handle resource request and release events, respectively. Step 3 decides whether or not deadlock exists in $\mathcal{O}(1)$ run-time. Step 4 computes reachable sink nodes for every resource. Steps 3 and 4 are the crucial steps of MDDA, which performs the computation for deadlock detection. During an invocation of MDDA, either Step 1 or 2 is firstly invoked according to the type of an event (either resource request or release). If the event is a resource request and the request is blocked, Step 3 is invoked to decide whether or not deadlock has

formed. If no deadlock is detected, Step 4 is invoked to update the information on reachable sink nodes for every resource. The updated information is to be used for deadlock detection in the next invocation of MDDA due to a blocked request event. That is, Step 3 needs to use results from Step 4. Thus, to prove that Step 3 detects deadlock correctly, we need to show that Step 4 has finished correctly computing all sink nodes for every resource in the first place. This proof of the correctness of Step 4 is critical to prove the correctness of MDDA.

Step 4 is a sequence of operations that takes a set of input values and produces another set of output values. Such an expected input/output relationship can be specified as a computational problem [4]. We formally define the computational problem solved by Step 4 of MDDA as follows:

Input: Sub-adjacency matrices $AR[]$ and $AG[]$.

Output: Matrix $Sink[]$

In the rest of this section, we firstly introduce and prove several lemmas and corollaries, which are useful in later proofs. Secondly, we show that Step 4 correctly computes all reachable sink nodes for every resource. Then, we use Theorems 4.1 and 4.2 to show that Step 3 detects deadlock correctly. These two parts together prove the correctness of MDDA. Lastly, based on the proof of the correctness of MDDA, we show that the run-time complexity of MDDA is $\mathcal{O}(min(m, n))$.

B. The Proof Methodology

Step 4 is the main computational procedure in MDDA. To prove the correctness of Step 4, we use a *loop invariant* approach [4], which is a commonly used approach to prove the correctness of an algorithm with a loop. The following shows the steps of such a proof that utilizes a loop invariant:

1) *Initialization*: It is true prior to the first iteration of the loop.
2) *Maintenance*: If it is true in the beginning of k^{th} iteration of the loop, it remains true at the end of k^{th} iteration.
3) *Termination*: When the loop terminates, the invariant provides a useful property that shows the correctness of the algorithm.

When (1) and (2) hold, the loop invariant is true prior to every iteration of the loop. Here proving (1) corresponds to the base case, and (2) corresponds to the inductive step. In (3), if we can show that the loop invariant leads to the desired output of the target computational problem, we have proved

the correctness of the algorithm. Note that the proof methodology using a loop invariant is similar to mathematical induction, where, to show that a statement is true, a base case and an inductive step are proved. Next, we first introduce relevant lemmas and corollaries, and then show how the *loop invariant* methodology is applied to the proof of the correctness of MDDA.

C. Preliminary Theorems

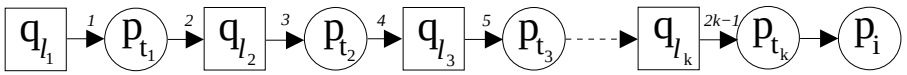
To make our proof more concise, we here introduce and prove several lemmas. Firstly, Lemma 6.1 through Corollary 6.6 that are used in the proof of the correctness of MDDA are described and proved. Secondly, Lemma 6.7 that is used in the proof of the run-time complexity of MDDA is proved.

Lemma 6.1: Any simple path from a resource node to a process node in a RAG consists of an even number of nodes.

Proof: We shall prove Lemma 6.1 by contradiction. Suppose the lemma has a counterexample, i.e., a simple path (say ε) that starts from a resource node (say q_{l_1}) and ends at a process node (say p_i), and consists of $2k+1$ nodes, $k{\geq}1$ (a path cannot consist of only one node). Let us assign an index i, $i{\geq}1$, to the edge that is encountered in the i^{th} place along path ε. An example of such a simple path is illustrated in Figure 8. Due to the bipartite property of a RAG, the first edge connects resource node q_{l_1} to a process node (say p_{t_1}). The second edge connects process node p_{t_1} to a resource node (say q_{l_2}). The third edge connects resource node q_{l_2} to a process node (say p_{t_2}). The fourth edge connects node p_{t_2} to a resource node (say q_{l_3}). Continuing this sequence, the $(2k-1)^{th}$ edge connects a resource node (say q_{l_k}) to a process node (say p_{t_k}). By Definition 3.3, all aforementioned nodes are distinct. Therefore, $2k$ distinct nodes out of $2k+1$ have appeared in path ε. Furthermore, p_i should not be included in the $2k$ nodes because if p_i do, path ε would have terminated and consist of less than $2k+1$ nodes. Thus, p_i is the only node excluded from the aforementioned $2k$ nodes. To complete path ε, p_i can only be connected to process node p_{t_k}. This is because if p_i is connected to any node other than p_{t_k}, ε is no longer a path by Definition 3.2. However, in a RAG, a process node cannot be connected to another process node. Therefore, the lemma has no counterexample; i.e., the lemma is true. ■

Corollary 6.2: Any simple path from a resource node to a process node in a RAG consists of an odd number of edges.

Proof: By Lemma 6.1, any simple path from resource q_j to process p_i has an even number of nodes. Thus, it consists of an odd number of edges. ■



Note: the number on each edge denotes the order of the edge in the path.

Figure 8. A simple path with $2k+1$ nodes.

Corollary 6.3: A path consisting of a simple path from a resource node to a process node and an edge from the ending process node to any resource node along the simple path contains an even number of edges.

Proof: By Corollary 6.2, a simple path (say ε) from a resource node (say q_j) to a process node (say p_i) has an odd number of edges (i.e., $2k-1$ edges, $k \geq 1$, where $2k$ is the number of nodes in the path). Hence, adding a new edge from p_i to any resource node in the simple path produces a new path (say ε') with a cycle. Since path ε' has one more edge than path ε, path ε' has $2k$ edges (i.e., an even number of edges). ■

An example of path ε' is shown in Figure 9.

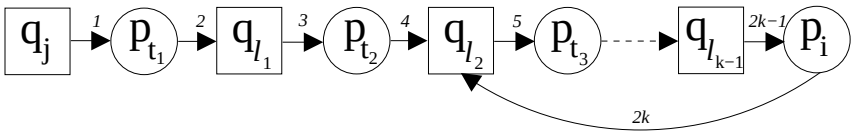


Figure 9. An example of path ε'.

Similar to Lemma 6.1 and Corollary 6.2, we have the following lemma and corollaries.

Lemma 6.4: Any simple path from a resource node to another resource node in a RAG consists of an odd number of nodes.

Proof: A simple path from a resource node (say q_j) to another resource node (say q_i) can be decomposed into an edge from a process node (say p_k) to q_i and a path from q_j to p_k. By Lemma 6.1, the path from q_j to p_k consists of an even number of nodes. Thus, the path from q_j to q_i consists of nodes that count an even number plus one. That is an odd number. Therefore, the lemma is true. ■

Corollary 6.5: Any simple path from a resource node to another resource node in a RAG consists of an even number of edges

Proof: By Lemma 6.4, any simple path from resource q_j to another resource q_i has an odd number of nodes. Thus, it consists of an even number of edges. ■

Corollary 6.6: A path consisting of a simple path from a resource node to another resource node and an edge from the ending resource node to any process node along the simple path contains an odd number of edges.

Proof: By Corollary 6.5, a simple path (say ε) from a resource node (say q_j) to another resource node (say q_i) has an even number of edges (i.e., $2k$ edges, $k \geq 1$, where $2k+1$ is the number of nodes in the path). Hence, adding a new edge from q_i to any process node in the simple path produces a new path (say ε') with a cycle. Since path ε' has one more edge than path ε, path ε' has $2k+1$ edges (i.e., an odd number of edges). ■

Having proved the lemmas and corollaries relevant to the proof of the correctness of MDDA, we now introduce and prove a lemma relevant to the proof of the run-time complexity of MDDA. Before the lemma is presented, let us examine Step 4 of Algorithm 5.3 more closely. Note that the number of iterations that Step 4 of Algorithm 5.3 executes for the search of all reachable sink nodes of resource q_j is determined by a longest path starting from q_j, not the sum of the numbers of edges for all paths. The reason is that although q_j and its reachable nodes can be connected via multiple paths with possible cycles, all paths are traversed independently and revisiting nodes in a cycle is prohibited in the algorithm. Thus, the upper bound on the number of iterations in Step 4 depends on the longest path possible that the algorithm may traverse.

Figure 10 illustrates a RAG containing multiple paths with cycles. In Figure 10, resource node q_{j_3} has two out-going edges, meaning two paths are directed from q_{j_3}. Thus, the algorithm traverses those two paths independently but in parallel. Note that one path becomes cycle φ_2. However, the algorithm only traverses every edge in φ_2 once per individual path. Similarly, two outgoing edges from q_{j_4}, one ending at p_{i_5} and the other ending at q_{i_3}, are traversed independently. In Figure 10, the longest path the algorithm will traverse is the path that starts from q_{j_1}, passes q_{j_2}, q_{j_3}, q_{j_4} as well as p_{i_4} and ends at q_{j_3} in cycle φ_1. Lemma 6.7 states an upper bound on the number of edges that the algorithm may traverse. Hence, Lemma 6.7 also signifies the maximum iterations that Step 4 takes to finish.

Lemma 6.7: An upper bound on the number of edges in any path that Step 4 in Algorithm 5.3 may traverse is $2 \times min(m, n)$, where m is the number of processes and n is the number of resources.

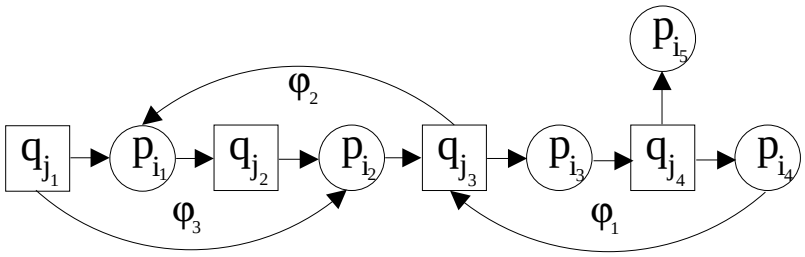


Figure 10. A RAG with multiple cycles and paths.

Proof: There exist two cases in RAGs: (i) $m \geq n$, (ii) $m < n$. For case (i), we consider two exclusive scenarios, a path with no cycle and a path with at least one cycle. Let us first consider a path that does not contain any cycle (i.e., a simple path). Starting from a resource node (say q_{j_1}), we can construct a simple longest path (say ε) (q_{j_1}, p_{i_1}), (p_{i_1}, q_{j_2}),...,$(p_{i_{n-1}}, q_{j_n})$,(q_{j_n}, p_{i_n}), where $\{j_1,\ldots,j_n\}$ is a permutation of $\{1,\ldots,n\}$ and $\{i_1,\ldots,i_n\}$ is a permutation of one of the subsets of $\{1,\ldots,m\}$, each of whose cardinality is n. Due to the bipartite property of a RAG, a path must consist of alternating process and resource nodes. Since all resource nodes appear in the aforementioned path, that path is one of the longest simple paths among all simple paths starting from resource q_{j_1} that one can construct. If only simple path ε is present, the maximum number of edges along a path is $2n-1$ according to Corollary 6.6. That longest path starts from q_{j_1} and ends at p_{i_n}. Now, let us consider inserting an edge into path ε aforementioned to create a cycle. A cycle (say φ) can be created by inserting a grant edge (q_{j_k}, p_{i_l}), where $2 \leq k \leq n$, $1 \leq l \leq n-1$, $k > l$, or request edge (p_{i_n}, q_{j_s}), where $1 \leq s \leq n$. In the case that a grant edge is inserted, varying k and l produces cycles consisting of different number of edges. The number of edges in a cycle varies from two edges (cycle φ_1 in Figure 11) to $2n-2$ edges (cycle φ_2 in Figure 11). However, such a grant edge does not alter the longest path. Therefore, when a grant edge is inserted, regardless of the number of edges contained in any cycle, the number of edges along the longest path is not affected and still $2n-1$.

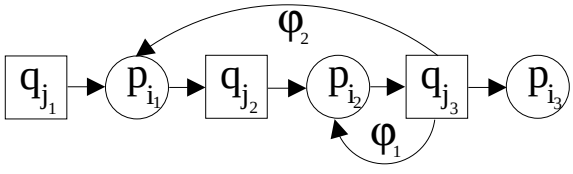


Figure 11. Cycles with $2n-1$ edges.

When request edge (p_{i_n}, q_{j_s}) is inserted, a cycle is formed (an example is shown in Figure 12). In any of these cases, the number of edges in the longest path becomes $2n$.

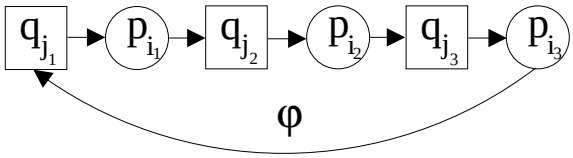


Figure 12. A cycle with $2n$ edges.

For cases where multiple cycles are present in a RAG as illustrated in Figure 10, because the algorithm traverses each path independently, the maximum number of edges that the algorithm may traverse depends solely on the longest path and thus is still $2n$. In summary, in case (i), the maximum number of edges the algorithm may traverse is $2n$.

Likewise, for case (ii), we first consider a scenario where only a simple path is present. We can construct a longest simple path ε' starting from q_{j_1} as (q_{j_1}, p_{i_1}),(p_{i_1}, q_{j_2}),...,$(p_{i_{m-1}}, q_{j_m})$,(q_{j_m}, p_{i_m}), $(p_{i_m}, q_{j_{m+1}})$, where $\{j_1,\ldots,j_m\}$ is a permutation of one of the subsets of $\{1,\ldots,n\}$, each of whose cardinality is m, and $\{i_1,\ldots,i_m\}$ is a permutation of $\{1,\ldots,m\}$. Since our systems are always in expedient states, resource node $q_{j_{m+1}}$ cannot be a sink node. Thus, $q_{j_{m+1}}$ should be connected to a process node among p_{i_1} through p_{i_m}. This forms a cycle, and such a cycle contains all process nodes and $m+1$ resource nodes. Using the same reasoning in case (i), we can infer that such a cycle is the longest path that the algorithm may traverse. Hence, in case(ii), the maximum number of edges that the algorithm may traverse is $2m$ (i.e., the number of edges in the simple path plus one more edge forming a cycle).

As a result, case (i) and (ii) show that the upper bound on the number of edges of any path that the algorithm may traverse is $2 \times min(m, n)$. ■

Note that the search for resource q_j's reachable sink nodes (the *while* loop in Step 4) starts from q_j, and then in each iteration the search visits process and resource nodes farther from q_j. From the proof of Lemma 6.7, we know that when a cycle is present in a RAG, Step 4 of Algorithm 5.3 visits each edge once, Thus, after Step 4 revisits the starting node of a cycle, it stops further traversing (i.e., revisiting other nodes in the cycle). Such a property of Step 4 of Algorithm 5.3 is the theoretical foundation that allows the algorithm to overcome cycles during the search, which is crucial for proving the correctness and run-time complexity of Step 4 in MDDA (i.e., Theorem 6.8). We conduct the proof of Theorem 6.8 using the *loop*

invariant approach. To better present our proof, we first measure the distance (see Definition 3.4) from the starting node of a search (i.e., q_j) to every node visited during the search. Then, the distance from q_j to any node visited in l^{th} iteration in Step 4 is always less than the distance from q_j to any node visited in k^{th} iteration if $k > l$. As seen in Step 4, when the search for q_j's reachable sink nodes completes in k^{th} iteration, its reachable sink nodes are found either in k^{th} iteration or in previous iterations, and are stored in the column vector $Sink[j][]^{(k)}$. Let us denote the distance from q_j to any of its reachable sink nodes that are found in k^{th} iteration as ζ. Thus, the distance of any q_j's reachable sink nodes stored in $Sink[j][]^{(k)}$ may be either ζ (i.e., found in k^{th} iteration) or less than ζ (i.e., found in any previous iteration). This way of describing the distance of any node in $Sink[j][]^{(k)}$ is illustrated in Figure 13, and utilized in the loop invariant for the proof of Theorem 6.8.

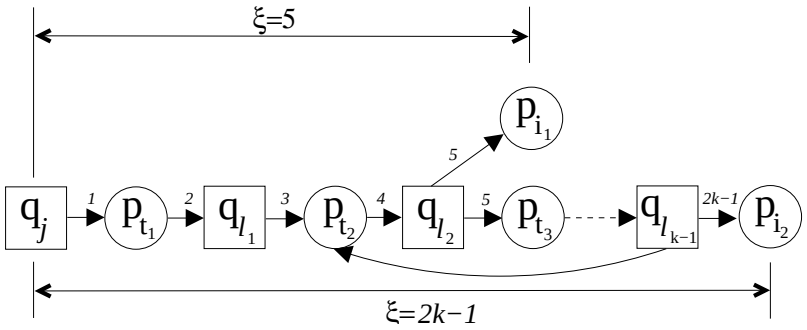


Figure 13. Illustration of the presentation of Sink[].

In Figure 13, both p_{i_1} and p_{i_2} are reachable sink nodes for q_j, with distances of 5 and $2k-1$, respectively. In k^{th} iteration, p_{i_2} is visited, and $Sink[j][]^{(k)}$ contains both p_{i_1} and p_{i_2}. Thus, any node denoted in $Sink[j][]^{(k)}$ has a distance from q_j that is either equal to $2k-1$ (i.e., for p_{i_2}) or less than $2k-1$ (e.g., for p_{i_1}). Note that $PrevResource[j][]^{(k)}$ is described in a similar way because a node denoted in $PrevResource[j][]^{(k)}$ may be visited either in k^{th} iteration or in any previous iteration.

Before we present Theorem 6.8 and its proof, an example is introduced to show the computations in one iteration of Step 4 in a particular scenario. This example illustrates the procedure how the search for reachable sink nodes traverses farther from resource nodes that have been visited in the last iteration. Such a procedure constitutes the induction step in our proof. Note that for each iteration, the search proceeds two edges (i.e., last resource→current process→current resource).

Example 4: Figure 14 is a RAG, whose corresponding sub-adjacency matrices $AR[]$ and $AG[]$ as well

as matrix $SinkProcess[]$ are as follows:

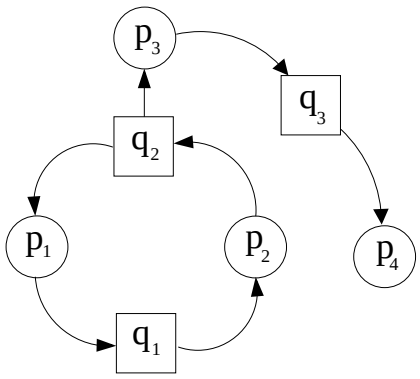


Figure 14. A RAG example.

$$\mathbf{AG_{3\times4}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{AR_{4\times3}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{SinkProcess_3} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

Next, we show the computations in the 1^{st} iteration of Step 4 in Algorithm 5.3 to illustrate an induction step. Before the 1^{st} iteration, all relevant matrix values are initialized in lines 22 to 25 of Algorithm 5.3. Their initialized values are:

$$\mathbf{WorkProcess^{(0)}_{3\times4}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{Sink^{(0)}_{3\times4}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{WorkResource^{(0)}_{3\times3}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{PrevResource^{(0)}_{3\times3}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Let us explain these initialized values relevant to q_1 in more detail. In the initialization stage, the search for all reachable sink nodes of q_1 has just started from q_1 itself, and has not visited any other nodes yet. Thus, $WorkProcess[1][]$ and $Sink[1][]$ contains all zeros, meaning that no process nodes have been visited by the search. $WorkResource[1][]$ is $[100]$, meaning that resource q_1 is being visited. $PrevResource[1][]$ represents all resource nodes that have been visited either in current iteration or in a previous iteration. Since there are no iterations before the initialization, $PrevResource[1][]$ is equal to $WorkResource[1][]$ (i.e., $[100]$).

Based on these initialized values, lines 27 to 30 compute the new matrix values as follows:

$$\mathbf{WorkProcess}^{(\mathbf{1})}_{\mathbf{3\times 4}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{Sink}^{(\mathbf{1})}_{\mathbf{3\times 4}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{WorkResource}^{(\mathbf{1})}_{\mathbf{3\times 3}} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{PrevResource}^{(\mathbf{1})}_{\mathbf{3\times 3}} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Now, let us examine lines 27 to 30 one by one. We take q_1 as an example and explain the computations relevant to q_1. Firstly, in line 27, $WorkProcess[1][]^{(1)}$ (i.e., the process nodes being visited in the current iteration) is computed (q_1 corresponds to the first column). In the RAG, q_1 has only one out-going edge directed to p_2 (i.e., $AG[1][2] = 1$). Therefore, $WorkProcess[1][]^{(1)}$ is $[0100]$, meaning that from q_1, p_2 is the only process node being visited in the 1^{st} iteration.

Secondly, in line 28, $Sink[1][]^{(1)}$ (corresponding to q_1) is computed. Until the 1^{st} iteration, from q_1 only one process node has been visited, which is p_2. Because p_2 is not a sink node, $Sink[1][]^{(1)}$ remains all zeros.

Thirdly, in line 29, $WorkResource[1][]^{(1)}$ is computed from $WorkProcess[1][]^{(1)}$. As is seen in the RAG, p_2 only has one out-going edge directed to q_2 (i.e., $AR[2][2] = 1$). Hence, $WorkResource[1][]^{(1)}$ is $[010]$, meaning that from q_1, q_2 is the only resource node being visited in the 1^{st} iteration.

Lastly, in line 30, $PrevResource[1][]^{(1)}$ is computed. Including both the initialization stage and the 1^{st} iteration, resource q_1 and q_2 have been visited. Thus, $PrevResource[1][]^{(1)}$ is $[110]$, representing both q_1 and q_2.

After lines 31 to 32 check the termination condition for the *while* loop (line 26), the 1^{st} iteration finishes. This iteration induces matrix values after the 1^{st} iteration (i.e., $WorkProcess[]^{(1)}$, $Sink[]^{(1)}$, $WorkResource[]^{(1)}$ and $PrevResource[]^{(1)}$) from their values before 1^{st} iteration (i.e., after the initialization). In the proof of Theorem 6.8, we will generalize the induction steps and prove that the induction procedure finally computes all reachable sink nodes for every resource correctly. ■

D. Proof of the Correctness of MDDA

Now, let us first prove the correctness of Step 4 of Algorithm 5.3 and then the correctness of MDDA.

Theorem 6.8: Step 4 in MDDA always terminates in a finite number of iterations and computes all reachable sink nodes for every resource upon its termination.

Proof: Those resources that have all units available have no edges connected in a RAG. Their corresponding columns in $Sink[]$ are merely assigned with zeros (line 22). Thus, Step 4 is trivially

correct for such resources. In this proof, let us focus on the computations (lines 24-32) for the rest of resources, each of which has at least one outgoing edge (i.e., their reachable sets are non-empty).

Note that $WorkResource[j][l]^{(k)}$ represents the value of $WorkResource[j][l]$ right after k^{th} iteration and thus also the value right before $(k+1)^{th}$ iteration. The same convention also applies to $PrevResource[j][l]$ and $Sink[j][i]$.

Now, let us state the loop invariant as follows. If the reachable set of q_j is non-empty, the followings hold: (i) For $k=0$, $1{\leq}j{\leq}n$, $1{\leq}l{\leq}n$, where n is the number of resources, $WorkResource[j][l]^{(0)}$ is 1 if l is equal to j; 0 otherwise. For $k=1$, $WorkResource[j][l]^{(1)}$ is 1 if q_l is reachable from q_j and the distance from q_j to q_l is 2; 0 otherwise. Similarly, for $k{\geq}2$, $WorkResource[j][l]^{(k)}$ is 1 if q_l is reachable from q_j and the distance from q_j to q_l is $2k$; 0 otherwise. (ii) For $k=0$, $1{\leq}j{\leq}n$, $1{\leq}l{\leq}n$, $PrevResource[j][l]^{(0)}$ is 1 if l is equal to j; 0 otherwise. For $k=1$, $PrevResource[j][l]^{(1)}$ is 1 if q_l is reachable from q_j and the distance from q_j to q_l is 0 or 2; 0 otherwise. Likewise, for $k{\geq}2$, $PrevResource[j][l]^{(k)}$ is 1 if q_l is reachable from q_j and the distance from q_j to q_l is less than (if visited previously) or equal to (if being visited currently) $2k$; 0 otherwise. (iii) For $k=0$, $1{\leq}i{\leq}m$, $1{\leq}j{\leq}n$, where m is the number of processes, $Sink[][]^{(0)}$ contains all zeros. For $k=1$, $Sink[j][i]^{(1)}$ is 1 if process p_i is a reachable sink node of q_j and the distance from q_j to p_i is equal to 1; 0 otherwise. Similarly, for $k{\geq}2$, $Sink[j][i]^{(k)}$ is 1 if process p_i is a reachable sink node of q_j and the distance from q_j to p_i is less than or equal to $2k-1$; 0 otherwise.

Next, let us show how the loop invariant holds in the phases of *initialization*, *maintenance* and *termination* for any resource q_j, where j is an arbitrary integer between 1 and n.

1) *Initialization*: We start by showing that the loop invariant holds when k=0. When k is 0, the loop invariant is trivially true because the search has not started yet and thus no reachable sink nodes for q_j have been found (i.e., $Sink[j][]^{(0)}$=$[0]_{1\times m}$). Also, the resource node of distance zero to q_j is itself (lines 24 and 25, i.e., for $1 \leq j \leq n$, $WorkResource[j][j]^{(0)}$=1, $PrevResource[j][j]^{(0)}$=1). Since $WorkResource[j][]^{(0)}$, $PrevResource[j][]^{(0)}$ and $Sink[j][]^{(0)}$ are consistent with the loop invariant, the base case of our induction (i.e., k=0) is established. The induction procedure that follows up is illustrated in Figure 15.
2) *Maintenance*: To see that each iteration maintains the loop invariant, let us first assume that the loop invariant holds at the beginning of λ^{th} iteration ($\lambda{\geq}1$), i.e., after $(\lambda{-}1)^{th}$ iteration. $WorkResource[j][l]^{(\lambda-1)}$ is 1 if resource q_l is reachable from q_j and its distance to q_j is $2\lambda{-}2$; 0 otherwise. $PrevResource[j][l]^{(\lambda-1)}$

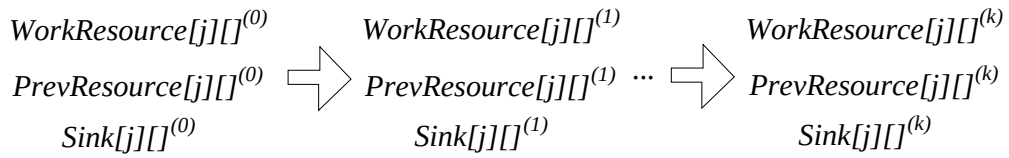


Figure 15. Illustration of the induction steps of Step 4.

is 1 if resource q_l is reachable from q_j and its distance to q_j is less than or equal to $2\lambda-2$; 0 otherwise. $Sink[j][i]^{(\lambda-1)}$ is 1 if process p_i is a reachable sink node of q_j and its distance to q_j is less than or equal to $2\lambda-3$; 0 otherwise. Note that when λ is 1, $Sink[j][i]^{(\lambda-1)}$ is 0 as defined in the loop invariant. Then, we need to prove the loop invariant holds at the end of λ^{th} iteration.

During iteration λ, $WorkProcess[j][i]^{(\lambda)}$ is first computed for all i, $1 \leq i \leq m$. In the rest of the iteration, $WorkProcess[j][i]^{(\lambda)}$ serves as an intermediate step for the calculations of $WorkResource[j][]^{(\lambda)}$ and $Sink[j][]^{(\lambda)}$. Let us look at how $WorkProcess[j][i]^{(\lambda)}$ is computed from $WorkResource[j][]^{(\lambda-1)}$ and $AG[][i]$ (line 27). By loop invariant assumption, $WorkResource[j][l]^{(\lambda-1)}$, for $1 \leq l \leq n$, is 1 if and only if reachable resource q_l's distance to q_j is $2\lambda - 2$. Meanwhile, $AG[l][i]$ is 1 if and only if an edge is directed from resource q_l to process p_i (i.e., $WorkProcess[j][i]^{(\lambda)}$ = 1). When both $WorkResource[j][l]^{(\lambda-1)}$ and $AG[l][i]$ are ones, we know that there exists a path from q_j to p_i via q_l. There can be multiple reachable resource nodes whose distance to q_j is $2\lambda - 2$ (i.e., $WorkResource[j][]^{(\lambda-1)}$ has multiple elements equal to one). Figure 16 illustrates such a scenario. If any of them has an edge directed to p_i, p_i is reachable from q_j (i.e., $WorkProcess[j][i]^{(\lambda)}$ = 1). So far, we have checked if a specific process node (p_i) is reachable from q_j. Beyond that, we need to know if any other processes among p_1 through p_m are reachable from q_j. In line 27, as i varies from 1 to m, it is checked in parallel whether any other processes are reachable from q_j by being connected to a reachable resource node whose distance to q_j is $2\lambda - 2$. That is, $WorkProcess[j][i]^{(\lambda)}$ = 1 for i = $1 \ldots m$.

After $WorkProcess[j][i]^{(\lambda)}$ for $1 \leq i \leq m$ are all computed, $Sink[j][]^{(\lambda)}$, $WorkResource[j][]^{(\lambda)}$ and $PrevResource[j][]^{(\lambda)}$ are computed in sequence as follows. In line 28, $Sink[j][i]^{(\lambda)}$ is computed. $Sink[j][i]^{(\lambda)}$ is 1 if it satisfies either of the two conditions: (i) p_i is a sink node and it is connected

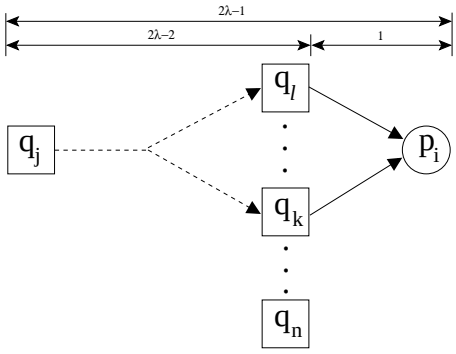


Figure 16. Illustration of the computation of $WorkProcess[j][i]^{(\lambda)}$.

to a reachable resource node whose distance to q_j is $2\lambda - 2$ (i.e., both $WorkProcess[j][i]^{(\lambda)}$ and $SinkProcess[i]$ are ones; an example is shown in Figure 16); (ii) p_i is a sink node which has been visited in a previous iteration (i.e., $Sink[j][i]^{(\lambda-1)}$ is 1). In the first case, p_i's distance to q_j is equal to $2\lambda - 1$; in the second case, its distance to q_j is less than $2\lambda - 1$. In summary, the loop invariant for $Sink[]$ holds at the end of λ^{th} iteration.

Now, let us look at the induction of $WorkResource[j][l]^{(\lambda)}$. As is afore-stated in this proof, $WorkProcess[j][i]^{(\lambda)}$ is 1 if and only if p_i is connected to some reachable resource node whose distance to q_j is $2\lambda - 2$ (i.e., path $q_j \rightarrow \ldots \rightarrow q_t \rightarrow p_i$ in Figure 17). Then, if $AR[i][l]$ is also 1, there exists an edge directed from p_i to q_l as seen in Figure 17. Thus, it indicates that there exists a path from q_j to q_l via p_i with 2λ edges as shown in Figure 17. Then, there are two cases: (i) q_l has never been visited in any previous iteration; (ii) q_l has been visited in a previous iteration. In case (i), $PrevResource[j][l]^{(\lambda-1)}$ is 0. Thus, the calculation in line 29 makes $WorkResource[j][l]^{(\lambda)}$ equal to 1. The loop invariant for $WorkResource[]$ holds because the distance from q_j to q_l is the number of edges in the current path, which is 2λ. In case (ii), because q_l has been visited in a previous iteration, $PrevResource[j][l]^{(\lambda-1)}$ is 1. Hence, q_l's distance to q_j is less than or equal to $2\lambda - 2$ according to the loop invariant. Thus, by the calculation in line 29, $WorkResource[j][l]^{(\lambda)}$ is 0. This signifies that if q_l's distance to q_j is not 2λ, $WorkResource[j][l]^{(\lambda)}$ is 0. In summary, we have proved that after iteration λ, $WorkResource[j][l]^{(\lambda)}$ is 1 if resource q_l's distance to q_j is 2λ (case i); 0 if resource q_l's distance to q_j is not 2λ (case ii). That is, if the loop invariant holds before iteration λ, it still holds after iteration λ. As a result, q_l will not be visited in any following iteration because in all following iterations $PrevResource[j][l]$

remains equal to 1.

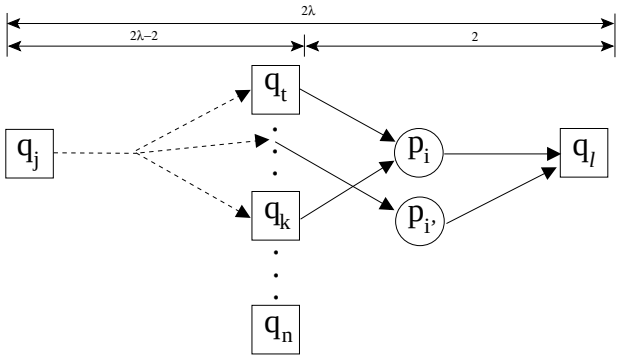


Figure 17. Illustration of the computation of $WorkResource[j][l]^{(\lambda)}$.

Lastly, $PrevResource[j][l]^{(\lambda)}$ becomes 1 if either $WorkResource[j][l]^{(\lambda)}$ or $PrevResource[j][l]^{(\lambda-1)}$ is 1. Note that if $PrevResource[j][l]^{(\lambda-1)}$ is 1, by the formula in line 29, $WorkResource[j][l]^{(\lambda)}$ must be 0. In the case $WorkResource[j][l]^{(\lambda)}$ is 1, $PrevResource[j][l]^{(\lambda)}$ becoming one signifies that resource q_l's distance to q_j is 2λ. In the case $PrevResource[j][l]^{(\lambda-1)}$ is 1, $PrevResource[j][l]^{(\lambda)}$ having become one signifies that resource q_l's distance to q_j is less than or equal to $2\lambda - 2$, thus certainly less than 2λ.

So far, we have proved that if the stated loop invariant holds before iteration λ, it must also hold after iteration λ.

3) *Termination*: Since the *while* loop will be terminated if the condition in line 31 holds, we first show that the condition in line 31 always comes to be in a certain iteration. Then, we show that upon the termination of the loop, the loop invariant holds and $Sink[]$ contains the sink bitmask for all reachable sink nodes for every resource.

Guaranteed termination: The goal of Step 4 is to find all reachable sink process nodes of every resource node. Because a RAG is a bipartite graph, every sink process node has at least one incoming edge from a resource node. Hence, when a sink process node (say p_s) is visited in iteration λ, a resource node that is directly connected to p_s should have been visited in iteration $\lambda - 1$. Therefore, when the

search for resource q_j's reachable sink nodes has visited all q_j's reachable resource nodes by iteration $\lambda-1$, it is assured that all q_j's reachable sink nodes will have been visited by iteration λ. To implement the aforementioned termination condition, the algorithm uses two matrices, i.e., $WorkResource$ and $PrevResource$, which contain newly and previously visited resource nodes, respectively. As the search for resource q_j's reachable sink nodes visits more resource nodes (note that the search traverses all paths starting from q_j and thus will visit all q_j's reachable nodes), eventually, $WorkResource[j][]$ will not contain any node and $PrevResource[j][]$ will contain all q_j's reachable resource nodes because all q_j's reachable resource nodes have been visited by that time. Thus, the search for resource q_j's reachable sink nodes terminates at the earliest time when it is found that all q_j's reachable resource nodes have been visited.

To prove that the loop invariant condition can produce the aforementioned termination condition, let us look at the meaning of the value of $PrevResource[j][l]$ at the end of iteration $\lambda-1$. From the *Maintenance* phase, we know that $PrevResource[j][l]^{(\lambda-1)}$ is 1 if the distance from q_l to q_j is less than or equal to $2\lambda-2$. Since our system contains only a finite number of processes plus resources, its corresponding RAG consists of a finite number of nodes. By Definition 3.3, any simple path in a RAG can only contain a finite number of nodes as well as edges because all nodes along a simple path are distinct. Hence, for any resource (say q_j), the maximum distance of any q_j's reachable resource node to q_j is a finite positive integer. Thus, as λ increases and becomes large enough, $2\lambda-2$ will become equal to the maximum distance of any q_j's reachable resource node to q_j. When that occurs, the loop invariant signifies that after iteration $\lambda-1$, $PrevResource[j][l]^{(\lambda-1)}$ is 1 if the distance from q_l to q_j is less than or equal to the maximum distance from any q_j's reachable resource node to q_j. Since the distance from any q_j's reachable resource node to q_j is less than or equal to the maximum distance, the loop invariant is equivalent to the fact that $PrevResource[j][l]^{(\lambda-1)}$ is 1 if q_l is a reachable resource node of q_j. Thus we can assure that all reachable resource nodes of q_j have been visited after iteration $\lambda-1$. If any of the newly visited resource nodes in iteration $\lambda-1$ is connected to a sink process node, that sink process node will be visited in iteration λ. For resource nodes visited in iteration λ_p where $\lambda_p<\lambda-1$, if any of them is connected with a sink process node (say p_s), p_s should have been visited in iteration λ_p+1, i.e., an iteration before iteration λ because $\lambda_p+1<\lambda$. Hence, we know that all reachable sink process nodes have been visited after iteration λ.

From the afore-stated discussion, we know that the termination iteration for q_j depends on the distance of the q_j's farthest reachable resource node to q_j. Even in case cycles exist in a RAG, by Lemma 6.7, we know that an upper bound on the number of edges in a path that the algorithm may traverse is still determined by the distance of the q_j's farthest reachable node to q_j. Considering how to compute the distance of the q_j's farthest reachable node to q_j, there are two cases: (i) The node farthest to q_j is one of its reachable process nodes (an example is process p_{i_3} in Figure 11). (ii) The node farthest to q_j is one of its reachable resource nodes (an example is resource q_{j_3} in Figure 10). In case (i), by Corollary 6.2, the distance of the q_j's farthest reachable node to q_j is a positive odd integer (say a). As a result, the distance of the q_j's farthest reachable resource node to q_j is $a-1$. When $2\lambda - 2 = a-1$, i.e., λ=$(a+1)/2$, $PrevResource[j][l]^{(\lambda-1)}$ must be 1 if q_l is reachable from q_j. This is because the distance from q_l to q_j must be less than or equal to $a-1$. Then, in iteration λ, where λ=$(a+1)/2$, all $WorkResource[j][l]^{(\lambda)}$ are zeros for $l = 1, \ldots, n$. Thus, the *while* loop terminates. In case (ii), by Corollary 6.5, the distance of the q_j's farthest reachable resource node to q_j is a positive even integer (say b). Likewise, when $2\lambda - 2 = b$, i.e., $\lambda = (b+2)/2$, $PrevResource[j][l]^{(\lambda-1)}$ must be 1 if q_l is reachable from q_j. Thus, the *while* loop terminates after iteration λ, where $\lambda = (b+2)/2$.

The loop invariant: For case (i) aforementioned in the part of *Guaranteed termination*, the *while* loop terminates after iteration λ, where $\lambda = (a+1)/2$. Due to *Maintenance* property, $Sink[j][i]^{(\lambda)}$ is 1 if p_i is a reachable sink node of q_j because the distance from p_i to q_j is always less than or equal to $2\lambda - 1$, which is exactly a (i.e., $2\times\frac{a+1}{2} - 1 = a$). As a result, all reachable sink nodes of q_j must have been found at this moment, which indicates that all paths from q_j to its reachable sink nodes have been searched. For case (ii), the *while* loop terminates after iteration λ, where $\lambda = (b+2)/2$. Since the node with the maximum distance b among all q_j's reachable nodes is a resource node, the maximum distance from q_j's reachable process node is $b-1$. Due to *Maintenance* property, $Sink[j][i]^{(\lambda)}$ is 1 if p_i is a reachable sink node of q_j because the distance from p_i to q_j is always less than or equal to $b-1$ and thus certainly less than $2\lambda - 1$ (i.e., $2\times\frac{b+2}{2} - 1 = b+1$). Similar to case (i), all paths from q_j to its reachable sink nodes have been searched. Thus, in either case, $Sink[j][]$ is a complete sink bitmask for all reachable sink nodes of q_j.

Combining *initialization*, *maintenance* and *termination*, we have proved that Step 4 of MDDA correctly computes all reachable sink nodes for resource q_j, where $1 \leq j \leq n$ and n is the number of resources. ■

Theorem 6.9: MDDA detects deadlock if and only if there exists a knot in the RAG.

Proof: By Theorem 6.8, Step 4 of MDDA always terminates and computes the reachable sink nodes for all resources correctly. It means that next time Step 3 is entered, $Sink[]$ consists of the sink bitmasks for all reachable sink nodes of all resources. Thus, if the equality in line 15 holds, $Sink[j][]$ (i.e., the sink bitmask for q_j) is the same as the bitmask of process p_i. By the definitions of the sink bitmask and the process bitmask (see Definitions 5.1 and 5.2), process p_i is the only reachable sink node of resource q_j (i.e., only $Sink[j][i]$ is 1). Hence, in such a scenario, by Theorem 4.2 a knot is formed in the corresponding RAG, and by Theorem 4.1 a deadlock occurs. The deadlock is revealed by the equality test in line 15. Thus, MDDA detects deadlock correctly. ■

E. Proof of the Run-time Complexity of MDDA

Lemma 6.10: When Step 4 of MDDA is parallelized so that the computations for all resources are performed simultaneously, Step 4 completes its computations in at most $min(m, n)$ iterations, where m and n are the number of processes and resources, respectively.

Proof: Since Step 4 is parallelized, it terminates when the computations for the resource requiring the maximum number of iterations are finished. Without loss of generality, let us assume that such a resource is q_j. By Lemma 6.7, an upper bound on the number of edges that Step 4 of Algorithm 5.3 traverses is $2 \times min(m, n)$. Because in each iteration Step 4 traverses two edges, the maximum number of iterations in Step 4 can be $2 \times min(m, n)/2$, i.e., $min(m, n)$ iterations. ■

Theorem 6.11: The run-time complexity of MDDA is $\mathcal{O}(min(m, n))$.

Proof: The run-time complexity of MDDA is determined by the run-time complexity of Steps 1 to 4. Since the run-time complexities of Steps 1 to 3 and Step 4 are $\mathcal{O}(1)$ and $\mathcal{O}(min(m, n))$, respectively, the overall run-time complexity of MDDA is $\mathcal{O}(min(m, n))$. ■

VII. HARDWARE IMPLEMENTATION

In this section, we delineate how Algorithm 5.3 is realized in hardware. The algorithm has been implemented using Verilog HDL.

A. Architecture of the MDDU

The simplified view of the architecture of a Multi-unit resource hardware Deadlock Detection Unit (MDDU) for three processes and three resources is shown in Figure 18. The architecture consists of *Matrix cells, SK/WP cells, WR/PR cells, SP cells, Free-unit cells* and a *Decision cell* in addition to *Finite State Machine cells* (*FSM cells* in short). In the following, we describe the architecture in detail.

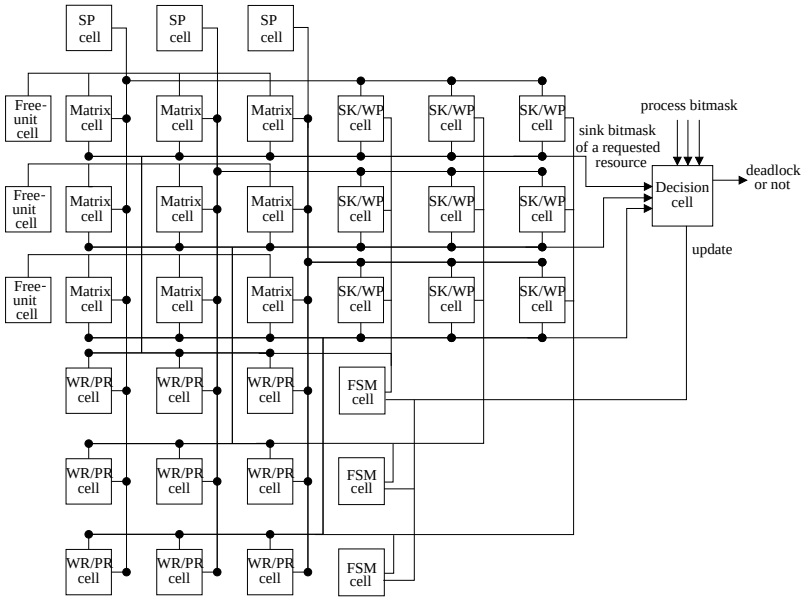


Figure 18. Architecture of 3x3 MDDU.

1) Matrix Cells, Free-unit Cells and SP Cells: The *Matrix Cell* at the i^{th} row and j^{th} column (denoted as MatrixCell[i][j]) contains three registers, which store the values of *AR[i][j], AG[j][i]* and Weight[j][i]. All the $m \times n$ *Matrix Cells* together store the complete resource allocation picture in the system. The contents of these cells are not altered during the computations of MDDA. Their contents can only be changed by the operating system when the resource allocation situation changes. Each *Free-unit cell* contains the value of the number of free units for each resource. When a unit of resource q_j is granted to process p_i, the value of AG[j][i] in the MatrixCell[i][j] is set to 1. The weight register in the same cell is increased by 1. Also, the value in the *Free-unit cell* on the j^{th} row is reduced by 1. When the request from process p_i for resource q_j is blocked, only the value of AR[i][j] in MatrixCell[i][j] is set to 1, and all other register values do not change. For the case where a unit of resource q_j held by process p_i is released, the weight register in MatrixCell[i][j] is decreased by 1, and the value in the *Free-unit cell*

on the j^{th} row is increased by 1. Information about whether or not a process node is a sink node in the RAG is stored in the *SP cells*. Each *SP cell* contains the value of an element in the vector SinkProcess[], which is computed by performing bit-wise OR operations among the elements of AR[i][] (i.e., the AR outputs of *Matrix cells* on the i^{th} row) as is described in Equation 1.

2) SK/WP Cells: The logic diagram of an *SK/WP cell* is shown in Figure 19. An *SK/WP cell* has two n-bit inputs as well as three 1-bit inputs and produces two 1-bit outputs. An *SK/WP cell* takes its inputs from a row of *WR/PR cells' WR* outputs and a column of *Matrix cells' AG* outputs in addition to an *SP cell* output. It computes its new values (i.e., WP[i][j] and SK[i][j]) according to Equations 2 and 3. It is important to note that lines 27 to 30 need to be computed in sequence. To achieve the sequential computation, an *SK/WP cell* has to be synchronized with the proper *WR/PR cells*. The input signals, namely *input_ready* and *done*, sent by *FSM cells* are used to synchronize the computations across proper cells and determine the correct termination point. The details will be explained when *FSM cells* are described.

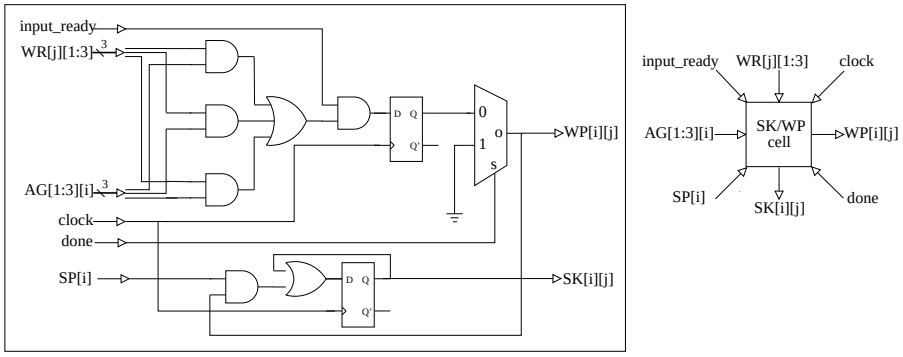


Figure 19. The logic diagram of an SK/WP cell.

3) WR/PR Cells: The logic diagram of a *WR/PR cell* is shown in Figure 20. A *WR/PR cell* has two m-bit inputs as well as two 1-bit inputs and produces one 1-bit output. A *WR/PR cell* takes its inputs from a column of *SK/WP cells' WP* outputs and a column of *Matrix cells' AR* outputs. It computes its new values (i.e., WR[j][k] and PR[j][k]) according to Equations 4 and 5. Note that the value of PR[j][k] is only used internally in a *WR/PR cell*. Thus, there is no output for PR[j][k]. Similar to an *SK/WP cell*, a *WR/PR cell* needs control signals from the *FSM cell* to synchronize and properly terminate its computations.

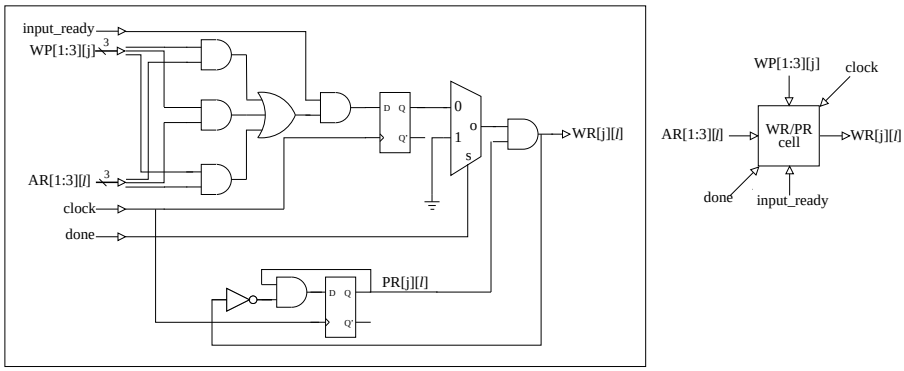


Figure 20. The logic diagram of a WR/PR cell.

4) Decision cell: The logic diagram of the *Decision cell* is shown in Figure 21. It performs deadlock detection by comparing the bitmask of the requester process with the sink bitmask for the reachable sink nodes of the requested resource (line 15 of Algorithm 5.3). If they are equal, a deadlock is detected. Afterwards, a deadlock recovery scheme would be brought in (it is out of the scope of this report). If they are different, the *Decision cell* sends out a signal to the *FSM cells*. The *FSM cells* then initialize necessary *SK/WP cells* and *WR/PR cells* and start updating the sink bitmasks for the sink nodes of all resources. In the next section, we explain how the *FSM cells* coordinate proper *SK/WP cells* and *WR/PR cells* to correctly perform the computations described in *Step 4* of Algorithm 5.3.

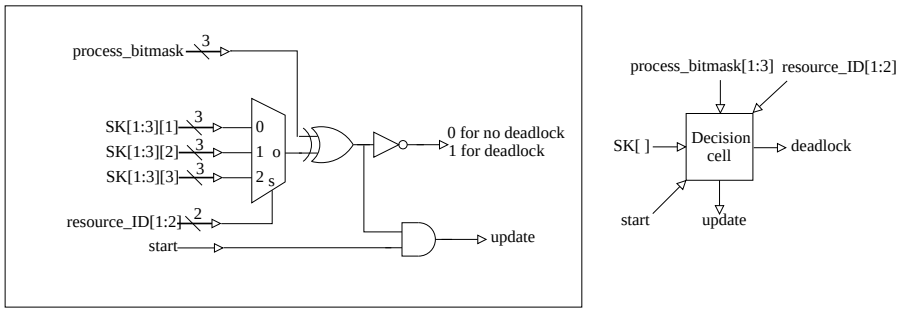


Figure 21. The logic diagram of the Decision cell.

5) FSM cells: The j^{th} row of *WR/PR cells* and the j^{th} column of *SK/WP cells* need to cooperate to perform and complete the computation of the sink bitmask for the reachable sink nodes of resource q_j. As described in lines 27 to 30 in Algorithm 5.3, line 27 uses the values of WorkResource[j][] (i.e., the values stored in the j^{th} row of *WR/PR cells*) from last iteration, while line 29 uses the values of

WorkProcess[][j] (i.e., the values stored in the j^{th} column of *SK/WP cells*) just computed in the same iteration. This implies that the computations of the j^{th} row of *WR/PR cells* and the j^{th} column of *SK/WP cells* have to be interleaved. The *FSM cells* are designed to serve such a purpose that either the j^{th} row of *WR/PR cells* or the j^{th} column of *SK/WP cells* are computing their new values at a time. The finite state machine implemented in an *FSM cell* is shown in Figure 22.

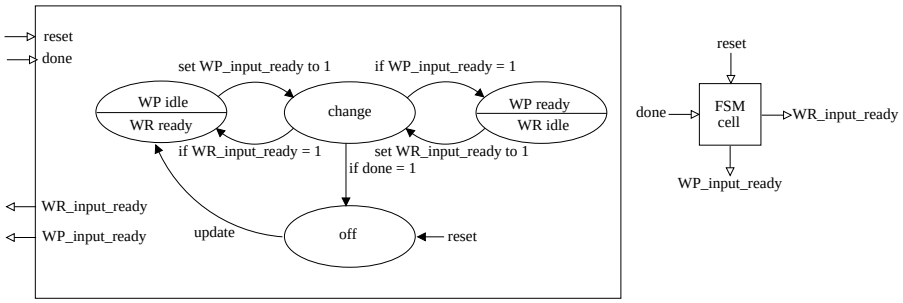


Figure 22. The state transition diagram of an FSM cell.

B. Synthesis of the MDDU

We use the Mentor Graphics ASIC Design Kit 3.0 [13] with the Taiwan Semiconductor Manufacturing Company .25 μm standard cell library [22] to synthesize MDDUs in different sizes. The synthesis results are shown in Table 4. The area of a synthesized MDDU is denoted by the number of units, each of which is equivalent to a minimum-sized two-input NAND gate in the library. In the case where an MPSoC contains one MDDU16x16, five PowerPC 755 processors (1.7M gates each [17]) and 256 MB DRAM memory (536M gates [8]), the resulting MPSoC area, the sum of the areas of five MPC775s plus 256 MB memory plus MDDU16x16 (i.e., 1.7M$\times$5+536M+130491), is 544,630,491 gates. Thus, the area overhead in the MPSoC due to the MDDU16x16, i.e., the area of MDDU16x16 divided by the total MPSoC area, is approximately .024 percent (i.e., 130,491/544,630,491).

Table 4. Synthesis Results of the MDDU

MDDU size	5x4	8x8	10x10	16x16	24x24
Area (w.r.t. 2-input NAND)	5509	23509	43683	130491	351778

VIII. Experiments and Their Results

A. Target System and Simulation Environment

Here we describe two sample target Multiprocessor System-on-a-Chips (MPSoCs) integrating our hardware Multi-unit resource Deadlock Detection Unit (MDDU). The architecture of the first target MPSoC is similar to Figure 7(a). It consists of five Motorola MPC755 processors, SRAM memory of 256MB and four multi-unit dummy resources. Each processor has one process running on it. The dummy resources count the specified time (set by the software) after their units are granted, and they interrupt the corresponding processor where a process uses the resource units when a preset time elapses. Then, the processor releases the dummy resource units. 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 second target system has the same configuration parameters as the first target except that the second system consists of four Motorola MPC755 processors with two processes running on each processor and eight dummy resource devices. The application is written in the C language and compiled using a PowerPC-GCC cross-compiler. We use Atalanta RTOS version 0.3 [21], a shared memory multiprocessor RTOS, to manage processes and resources. In order to simulate such MPSoCs with the multiprocessor operating system, we use Mentor Graphics Seamless hardware-software Co-Verification Environment (CVE) [14] aided by ModelSim [15] for hardware simulation and XRAY for software debugging.

B. A 5x4 Application Scenario

We test the 5x4 MDDU with an application consisting of five processes each running on a processor of the first target system. The sequence of events in Table 5 gradually builds the system status described by the RAG in Figure 7(b). Events e_{11} through e_{13} are presented in Section V-D for the detailed explanation of our algorithm. Specifically, after event e_{11} and before e_{12}, the system status is the same as described in Figure 7(b). Furthermore, e_{12} changes the status of the system such that the $sink$ nodes for every resource is the same as shown in Table 3. Then, e_{13} incurs deadlock at t_{13}. Since the system state after t_{13} is affected by the deadlock recovery scheme employed, we stopped our deadlock detection experiment after t_{13}. Deadlock detection always takes two clock cycles for all resource events listed in Table 5. We are particularly interested in detection preparation time since it is the dominant factor in the overall run-time complexity. We specifically count the number of clock cycles elapsed from the moment the *Decision cell*

Table 5. An Application Scenario in the 5x4 Target System

Time	Event	System Operation	Preparation Clock Cycles
t_1	e_1	MPC755-4 requests resource 1; resource 1 is granted to MPC755-4 immediately.	2.5
t_2	e_2	MPC755-1 requests resource 3; resource 3 is granted to MPC755-1 immediately.	1.5
t_3	e_3	MPC755-4 requests resource 1; resource 1 is granted to MPC755-4 immediately.	1.5
t_4	e_4	MPC755-5 requests resource 4; resource 4 is granted to MPC755-5 immediately.	1.5
t_5	e_5	MPC755-3 requests resource 2; resource 2 is granted to MPC755-3 immediately.	1.5
t_6	e_6	MPC755-2 requests resource 2; resource 2 is granted to MPC755-2 immediately.	1.5
t_7	e_7	MPC755-4 requests resource 2; this request is blocked, no deadlock.	5.5
t_8	e_8	MPC755-1 requests resource 1; resource 1 is granted to MPC755-1 immediately.	5.5
t_9	e_9	MPC755-5 requests resource 1; resource 1 is granted to MPC755-5 immediately.	5.5
t_{10}	e_{10}	MPC755-2 requests resource 3; this request is blocked, no deadlock.	9.5
t_{11}	e_{11}	MPC755-3 requests resource 1; this request is blocked, no deadlock.	9.5
t_{12}	e_{12}	MPC755-1 requests resource 4; this request is blocked, no deadlock.	9.5
t_{13}	e_{13}	MPC755-5 requests resource 2; this request is blocked, **deadlock**.	N/A

sends out the "update" signal to the moment the last FSM cell terminates the computations in the *WR/PR cells* and the *SK/WP cells* (i.e., all WR[] elements become zeros). The results for detection preparation time are also listed in Table 5.

Typically, each iteration in *Step 4* includes the sequential computations from Equations 2 through 5. However, in the first iteration, the new values of WP are always the same as the values in AG^T because the process nodes that are directly connected to each resource are already stored in *AG* as the grant edges in the RAG (see AG and WP after the 1^{st} iteration in Section V-D). Therefore, in the first iteration, we can skip the computation of matrix *WP* (Equation 2). As a result, the first iteration only takes 1.5 clock cycles while other iterations all take 4 cycles. In all successive iterations, *SK/WP cells* and *WR/PR cells* operate under the control of the FSM depicted in Figure 22. Therefore, in each of the 2^{nd} and later iterations, four transitions occur in the FSM (i.e., either "WP idle/WR ready"⇒"change"⇒"WP ready/WR idle"⇒"change"⇒"WP idle/WR ready" for non-terminating iterations, or "WP idle/WR ready"⇒"change"⇒"WP ready/WR idle"⇒"change"⇒"off" for the terminating iteration). Hence, each of such iterations takes four clock cycles.

Our simulation demonstrates that the MDDU takes 9.5 clock cycles to compute the sink bitmasks for all resources in the RAG shown in Figure 7(b). That is equivalent to three iterations of the *while* loop in *Step 4*. In our 5x4 simulated target, the maximum number of iterations in *Step 4* can be four because of the afore-stated $\mathcal{O}(min(m,n))$ run-time complexity where *m = 5* and *n = 4*.

C. An 8x8 Application Scenario

We also simulated a separate 8x8 application scenario on the second target system with an 8x8 MDDU. Although the second target system is larger than the the first one, deadlock detection still takes only two cycles for all resource events. In the same way, we measured the detection preparation clock cycles for all resource events. All simulated resource events and their corresponding detection preparation times are listed in Table 6. In this 8x8 simulated target system, the maximum number of iterations in *Step 4* can be eight. In the application scenario shown in Table 6, the longest preparation took 13.5 clock cycles, which is equivalent to four iterations in *Step 4*.

Table 6. An Application Scenario in the 8x8 Target System

Time	Event	System Operation	Preparation Clock Cycles
t_1	e_1	process 1 requests resource 1; resource 1 is granted to process 1 immediately.	2.5
t_2	e_2	process 2 requests resource 2; resource 2 is granted to process 2 immediately.	1.5
t_3	e_3	process 6 requests resource 4; resource 4 is granted to process 6 immediately.	1.5
t_4	e_4	process 3 requests resource 3; resource 3 is granted to process 3 immediately.	1.5
t_5	e_5	process 7 requests resource 8; resource 8 is granted to process 7 immediately.	1.5
t_6	e_6	process 2 requests resource 1; resource 1 is granted to process 2 immediately.	1.5
t_7	e_7	process 5 requests resource 3; resource 3 is granted to process 5 immediately.	1.5
t_8	e_8	process 6 requests resource 7; resource 7 is granted to process 6 immediately.	1.5
t_9	e_9	process 3 requests resource 2; resource 2 is granted to process 3 immediately.	1.5
t_{10}	e_{10}	process 4 requests resource 1; resource 1 is granted to process 4 immediately.	1.5
t_{11}	e_{11}	process 4 requests resource 6; resource 6 is granted to process 4 immediately.	1.5
t_{12}	e_{12}	process 3 requests resource 4; this request is blocked, no deadlock.	5.5
t_{13}	e_{13}	process 7 requests resource 6; resource 6 is granted to process 7 immediately.	5.5
t_{14}	e_{14}	process 5 requests resource 5; resource 5 is granted to process 5 immediately.	5.5
t_{15}	e_{15}	process 7 requests resource 7; resource 7 is granted to process 7 immediately.	5.5
t_{16}	e_{16}	process 8 requests resource 6; resource 6 is granted to process 8 immediately.	5.5
t_{17}	e_{17}	process 4 requests resource 5; resource 5 is granted to process 4 immediately.	5.5
t_{18}	e_{18}	process 6 requests resource 6; this request is blocked, no deadlock.	13.5
t_{19}	e_{19}	process 7 requests resource 4; this request is blocked, no deadlock.	13.5
t_{20}	e_{20}	process 8 requests resource 4; this request is blocked, no deadlock.	13.5
t_{21}	e_{21}	process 5 requests resource 7; this request is blocked, **deadlock**.	N/A

IX. Discussion and Conclusion

Since both of Shiu's algorithm and ours have the same $\mathcal{O}(min(m,n))$ overall run-time complexity, we here show the comparison on various aspects between Shiu's algorithm versus ours and highlight the differences and the novel features of ours. A summary of the comparison is shown in Table 7. 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 other aspects.

In this report, firstly a novel $\mathcal{O}(1)$ parallel Multi-unit resource Deadlock Detection Algorithm (MDDA) is presented. The described algorithm is designed to be fully hardware-oriented and exploits the maximum

Table 7. The Comparison between Our Algorithm and Shiu's Algorithm [18]

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	$\mathcal{O}(min(m,n))$	$\mathcal{O}(min(m,n))$

computational parallelism in detection preparation by hardware. The algorithm has a run-time complexity of $\mathcal{O}(1)$ for detecting deadlock and $\mathcal{O}(min(m,n))$ for preparing detection. Secondly, the proofs of the correctness and run-time complexity of MDDA using the *loop invariant* method (a mathematical induction alike approach) are provided. Lastly, we discuss the implementation of MDDA in hardware (MDDU).

The MDDU provides very fast and deterministic run-time detection of deadlocks for both multi-unit as well as single-unit resource systems. It is especially beneficial for Multiprocessor System-on-Chip (MPSoC), which we predict will prevail as embedded system platforms. The MDDU stores the Resource Allocation Graph (RAG) of a system in a set of matrices. Whenever a resource request is blocked, the MDDU makes deadlock decision right away by comparing the information on the reachable sink nodes of the requested resource with the ID of the requester process node in a RAG. If no deadlock is detected, the MDDU updates the reachable sink nodes of all resources in parallel (detection preparation) to prepare for the next detection invocation. Compared with Kim's algorithm [9], the deadlock preparation run-time complexity is significantly reduced from $\mathcal{O}(n \times m)$ to $\mathcal{O}(min(m,n))$.

Through our experiments, we demonstrate the following: i) regardless of the size of the system (i.e., m and n), the MDDU always takes only two clock cycles to make its deadlock decision, and ii) when we increase the size of system from 5x4 to 8x8, the worst case run-time does not increase quadratically, but merely doubles. Finally, the MPSoC area overhead due to MDDU is small, approximately 0.024 percent for MDDU16x16 in our example MPSoC.

X. Acknowledgments

This research has utilized software and hardware donations from Sun and Mentor Graphics. Atalanta RTOS was developed at the Georgia Institute of Technology. We would like to thank IUPUI RSFG internal grant that enabled the research performed here.

References

[1] I. Cahit, "Deadlock detection using (0,1)-labeling of resource allocation graphs," In *IEE Proc of Computers and Digital Techniques*, pages 68–72, 1998.

[2] Chip multiprocessor, http://en.wikipedia.org/wiki/Multi-core_%28computing%29, visited July 2007.

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

[4] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms, Chapter 22*. The MIT Press, Cambridge , Massachusetts, 2nd edition, 2001.

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

[6] IBM, Cell broadband engine. http://www.research.ibm.com/cell, visited July 2007.

[7] IBM Tokyo Research Lab, Embedded systems. http://www.research.ibm.com/trl/projects/embedded_e.htm, visited July 2007.

[8] K. Itoh, *VLSI Memory Chip Design*. Springer-Verlag, 2001.

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

[10] 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.

[11] 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.

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

[13] Mentor Graphics, ASIC Design Kit. http://www.mentor.com/company/higher_ed/asic/index.cfm, visited July 2007.

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

[15] Mentor Graphics, Modelsim. http://www.mentor.com/modelsim, visited in May 2007.

[16] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, 38(8):114–117, 1965.

[17] MPC755 RISC Microprocessor Hardware Specifications, http://www.freescale.com/files/32bit/doc/data_sheet/MPC755EC.pdf, 2006.

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

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

[20] A. Silberschatz and P. Galvin, *Operating System Concepts*. John Wiley & Sons, 1999.

[21] 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.

[22] Taiwan Semiconductor Manufacturing Company, http://www.tsmc.com/, visited July 2007.

[23] Tilera Corp. Tile64 processor family. http://www.tilera.com/products/processors.php, visited in Jan. 2008.

[24] D. Wentzlaff et al. "On-chip interconnection architecture of the tile processor," *IEEE Micro*, 27(5):15–31, 2007.