AN *O(MIN(M,N))* PARALLEL DEADLOCK DETECTION ALGORITHM

JAEHWAN LEE AND VINCENT JOHN MOONEY III

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

GEORGIA INSTITUTE OF TECHNOLOGY

{jaehwan, mooney}@ece.gatech.edu

Abstract This paper presents a novel Parallel Deadlock Detection Algorithm (PDDA) and its hardware implementation, Deadlock Detection Unit (DDU). PDDA uses simple boolean representations of request, grant and no activity so that the hardware implementation of PDDA becomes easier and operates faster. We prove that the DDU has a worst-case run-time of $2 \times min(m,n) - 3 = O(min(m,n))$, where m is the number of resources and n is the number of processes. Previous algorithms in software, by contrast, have $O(m \times n)$ run-time complexity. We also prove the correctness of PDDA and the DDU. The DDU *reduces* deadlock detection time *by 99.9%, (i.e., 1000X)* or more compared to software implementations of deadlock detection algorithms. An experiment involving a practical situation that employs the DDU showed that the time measured from application initialization to deadlock detection was *reduced by 46%* compared to detecting deadlock in software.

1. Introduction

The users of a real-time operating system (RTOS) wish that it would provide a predictable response time at an affordable cost. To fulfill this wish, many researchers have investigated various approaches that ensure RTOS predictability. One active approach is to utilize a hardware mechanism since (i) hardware is typically far more predictable than software implementation of the same algorithm and since (ii) the cost of hardware decreases dramatically in accordance with Moore's law (or, more accurately, Moore's prediction) [1]. To help meet the predictability requirements of real-time embedded System-on-a-Chip (SoC) programmers, this paper presents a solution to deadlock detection.

Putting deadlock detection in hardware is extremely important since any request for or grant of a resource might result in deadlock. Invoking software deadlock detection on every resource allocation event would cost too much compute power; thus, using a software implementation of deadlock detection would perhaps be impractical in terms of the performance cost. A promising way of enabling deadlock detection with small compute power is to implement deadlock detection in hardware. In fact, without the aid of hardware, deadlock checking and meeting every other deadline would be unthinkable for most real-time embedded systems that need to detect deadlock. Practical systems may potentially initiate deadlock recovery more quickly and save large investments. For example, Mars Pathfinder, a real-time robot, was resetting on some priority inversion condition by downloading a new code over radio. By quickly detecting such a situation and calling a special "*recover*" boot code (which is more proven and has no deadlock), accidents can be avoided [23].

This paper presents, in detail, a novel parallel deadlock detection algorithm, which we call PDDA, and its hardware implementation, Deadlock Detection Unit (DDU) [11]. This paper also proves the correctness of PDDA and the run-time complexity of the DDU. However, the problem of deadlock recovery (i.e., what to do once deadlock is detected) is not within the scope of this paper, which focuses only on deadlock detection.

Example 1. Deadlock
In a system-on-a-chip (SoC) application, a processor may have to use several resources, for example, to process streaming data. An example of such a system shown in Figure 1 has two processors, a digital signal processor processing (DSP) and a video signal processor (VSP), and two resources, a wireless interface (WI) and an image coprocessor (ImC). Each processor (DSP or VSP) has to use both resources exclusively to complete its processing of the streaming data. In the case shown in Figure 1, the DSP holds resource ImC while the VSP holds resource WI. Furthermore, the DSP requests the WI, and the VSP requests the ImC; thus, a deadlock occurs since neither the DSP nor the VSP gives up or releases the resources they currently hold; instead, they wait for their resource requests to be fulfilled.

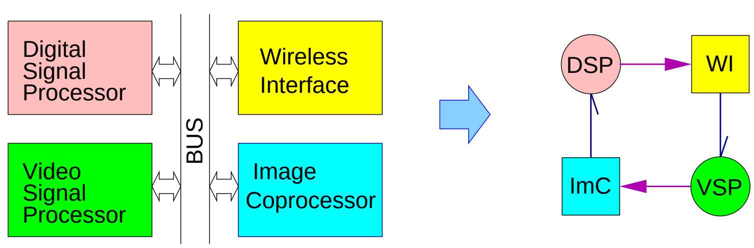


FIGURE 1. A deadlock example

□

Since the 1960's, operating system research has extensively focused on deadlock in computer systems [2][4][19][20][21][22]. A definition of deadlock can be written as follows.

Definition 1. *A system has a deadlock if and only if the system has a set of processes (tasks), each of which is blocked (either preempted or spinning), waiting for requirements that can never be satisfied.*

The following necessary deadlock conditions have been stated in some form or another in previous work; nonetheless, for clarity, we state the following five conditions which, if present, indicate that a system has a deadlock.

Condition 1. *Mutual exclusion – resources cannot be shared.*

Condition 2. *No preemption – a resource can only be released by the process holding it.*

Condition 3. *Partial allocation – a process holding resource(s) can request additional resources.*

Condition 4. *Resource waiting – a process must wait for all requested unavailable resources to become available before proceeding.*

Condition 5. *Circular hold and wait – a closed chain of alternate processes and resources exists such that each process holds at least one resource needed by the next process in the chain, and no process can complete without receiving all the requested resource(s).*

If all five of these conditions hold, then such a system has a deadlock. Please note that some conditions need not be strictly true at all times, but instead must be true only for the time(s) at which the system is deadlocked. For example, most modern chips have a "reset" pin that resets the entire chip, causing all the processes running on the chip's computational circuitry to release all held resources. Obviously, utilizing such a reset pin clears any current deadlocks by breaking Condition 2.

The remainder of this paper is organized as follows. Section 2 presents the motivation for this research, and Section 3 provides a comparative analysis of our work and previous work related to this research. Section 4 introduces the basic definitions and properties of a resource allocation graph (RAG) and deadlock while Section 5 provides a model of our system and some underlying assumptions. Section 6 introduces our approach in PDDA, shows basic deadlock theorems along with their proofs established on top of our assumptions, proposes a translation of a system state from a RAG into a matrix, and then presents PDDA in detail. Section 7 illustrates a hardware implementation of PDDA, and Section 8 describes the experimental setup and results of (i) an algorithmic execution-time comparison with various numbers of request and grant edges and (ii) an application execution-time comparison between the hardware implementation and a software implementation of an $O(m \times n)$ algorithm with five processes and five resources. Section 9, the final section, concludes the paper.

2. **Motivation**

Current trends show that System-on-a-Chip (SoC) technology has contributed to a significant evolution in digital chip design. Unlike a printed-circuit-board (PCB) filled with many digital chips, an SoC is designed as a hardware platform that integrates most of the functions of the end product in a single chip. An SoC commonly incorporates at least one processing element (PE) (e.g., a microprocessor or a DSP) or more that run embedded software. An SoC may include peripherals, reconfigurable logic and interfaces to the outside world, and it employs a bus-based architecture. An SoC may also contain both memory and analog functions. Current and future SoC technology will facilitate the creation of complex digital systems that are small, portable, energy efficient and reliable. Some examples of such complex digital systems are miniature personal digital assistants (PDAs) and digital cameras.

According to the current trends, we predict that many data streaming applications with many processes and resources will likely run on a single SoC in the near future. Thus, more and more interactions among processes and resources occur. Moreover, many applications consist of processes that require exclusive access not to one resource, but to several resources, which will increase the likelihood of deadlock. Thus, we believe that the demand for deadlock detection in an SoC will rise significantly.

To exemplify the current trends more specifically, consider the following realistic application. Suppose a future handheld device with SoC technology as shown in Figure 2 is developed and equipped with multiple processing elements (PEs) and resources. The device can capture motion pictures as well as still images, process Motion Picture Experts Group (MPEG) encoding and decoding, handle Joint Photographic Experts Group (JPEG) encoding and decoding, resize for e-mail, reformat for thumbnail printing, and perform digital signal processing such as noise reduction and picture enhancement. The device can also support data streaming applications using a standard wireless Local Area Network (LAN) card. In this environment, such a device will have many resources and processes (due to the future trend of many multimedia streaming applications that this device will support), which will likely result in multiple resource requests and grants, causing the system to experience some deadlock situations. Responding to the possible demand for deadlock detection in such a system and ensuring predictability in SoC design require a fast and predictable deadlock detection method. Such a method is proposed in this paper.

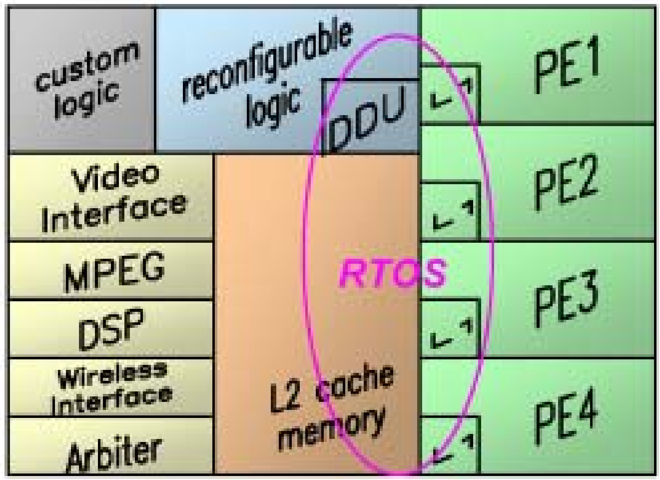


FIGURE 2. A practical SoC example drawn from a future handheld device

3. Previous Work

Ideally, application programmers would implement their applications in a way that guarantees that no deadlock would occur. Deadlock can be prevented by designing the resource allocation method of an RTOS such that one of Conditions 1-5 in Section 1 cannot occur. Some typical methods are (i) the ordered-requests method, in which resources are globally ordered [7], and (ii) the collective-requests method, in which each process requests all its necessary resources at once, i.e., no incremental requests [7]. However, these methods have associated performance penalties. Furthermore, an unforeseen race condition could occur, resulting in what was thought be deadlock free code reaching a deadlock. Therefore, if the performance penalty due to allocating resources in a way guaranteed not to result in deadlock is too high, or if the application programmer's manager does not believe that the application is really deadlock free, then a very fast and efficient way to use hardware to detect deadlock could be helpful, which is the focus of this paper: fast deadlock detection in parallel hardware logic.

In 1970, Shoshani et al. proposed an $O(m \times n^2)$ run-time complexity detection algorithm, based on a state graph model of resource allocation [3]. About two years later, Holt proposed an $O(m \times n)$ algorithm that introduces a graph reduction method to detect a knot that tells whether deadlock exists or not [4][18]. Holt's model of multiple processes and resources provides a versatile representation of resource allocation, and his approach describes a general resource system that models consummable as well as reusable resource types. Leibfried proposed a method of describing a system state using an adjacency matrix representation and a corresponding scheme that detects deadlock with matrix multiplications but with a run-time complexity of $O(m^3)$ [5]. Kim and Koh proposed an algorithm with $O(m \times n)$ time for deadlock detection preparation; thus an overall time for detecting deadlock (starting from a system description that just came into existence, e.g., due to multiple grants and requests occurring within a particular time or clock cycle) of at least $O(m \times n)$ [6]. In addition, Kim's approach is based on linked lists.

All of the above algorithms, however, assume an execution paradigm of one instruction or operation at a time. With a custom hardware implementation of a deadlock algorithm, parallelism can be exploited. To handle this possibility, PDDA and the DDU have recently been proposed [11]. However, Shiu et al. [11] did not formally prove the correctness and run-time complexity of the proposed approach. In this paper, we prove that (i) PDDA detects all and only authentic deadlocks and that (ii) the DDU has a run-time complexity of $O(min(m, n))$. The DDU operates on a matrix consisting of a simple boolean representation of the types of each edge: the request edge of a process requesting a resource, the grant edge of a resource granted to a process, or no activity (neither a request nor a grant). Not only that, by implementing PDDA with a small amount of hardware, the designed deadlock detection unit hardly affects system performance (and potentially has no negative impact whatsoever) yet provides the basis for enhanced deadlock detection applications.

PDDA distinguishes itself from others in that PDDA deals with edges that are not involved in cycle(s) as opposed to other algorithms that try to find an exact, specific deadlock cycle, which typically requires more computation time. In addition, PDDA traces neither cycles nor paths, and it does not require linked lists either. Not only that, by implementing PDDA with a small amount of hardware, the designed deadlock detection unit hardly affects system performance (and potentially has no negative impact whatsoever) yet provides the basis for enhanced deadlock detection applications.

4. **Basic Definitions**

Before presenting PDDA in detail and proving its correctness as well as its run-time complexity, we would like to introduce some basic definitions in order to effectively deliver our methodology. Specifically, we will introduce a few terms and definitions from graph theory and examine how to represent the deadlock problem with a RAG.

4.1. **Definition of a Resource Allocation Graph**

Definition 2. *Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n requesters or processes that may request and/or hold a number of resources at any given time.*

Definition 3. *Let* $Q = \{q_1, q_2, \ldots, q_m\}$ *be a set of* m *resources that provide specific functions usable by the processes.*

Definition 4. *Let the set of nodes* V *be* $\{P \cup Q\}$*, which is divided into two disjoint subsets* P *and* Q *such that* $P \cap Q = \emptyset$*. We also use another notation for the set of nodes,* $V = \{v_1, v_2, \ldots, v_l\}$*.*

Definition 5. *Let* G *be a set of grant edges. Let an ordered pair* (q_i, p_j) *be a grant edge where the first node is a resource* $q_i \in Q$ *and the second node is a process* $p_j \in P$ *and where* q_i *has been granted to* p_j*. Thus, a set of grant edges* G *can be written as* $G = \Big\{(q_i, p_j)$*, such that* $i \in \{1, 2, 3, \ldots, m\}$*,* $j \in \{1, 2, 3, \ldots, n\}$*, and resource* q_i *has been granted to process* $p_j\Big\}$*. An ordered pair* (q_i, p_j) *can also be represented by* $q_i \rightharpoonup p_j$ *or simply* $g_{i \rightharpoonup j}$*, where the harpoon "*$\rightharpoonup$*" represents a grant edge.*

Definition 6. *Let* R *be the set of request edges. Let an ordered pair* (p_j, q_i) *be a request edge where the first node is a process* $p_j \in P$*, and the second node is a resource* $q_i \in Q$ *and where* p_j *has requested* q_i *but has not yet acquired it. Thus, a set of request edges* R *can be written as* $R = \Big\{(p_j, q_i) | j \in \{1, 2, 3, \ldots, n\}, i \in \{1, 2, 3, \ldots, m\}$*, and process* p_j *is requesting resource* $q_i\Big\}$*. An ordered pair* (p_j, q_i) *can also be represented by* $p_j \rightarrow q_i$ *or simply* $r_{j \rightarrow i}$*, where the arrow "*$\rightarrow$*" represents a request edge.*

Definition 7. *Let the edge set* E *be* $\{R \cup G\}$*. We also use another notation,* $E = \{e_1, e_2, \ldots, e_h\}$*.*

Definition 8. *A given system with processes and resources can be abstracted by a Resource Allocation Graph (**RAG**). A RAG is a directed graph* $\gamma = \{V, E\}$*, such that* V *is a non-empty set of nodes defined in Definition 4, and* E *is a set of ordered pairs of edges*[1] *defined in Definition 7.*

Note that the edge set E of a particular RAG may be empty at a moment when processes neither have any outstanding requests nor hold any resources.

Definition 9. *Given RAG* γ*, let function* $E(\gamma)$ *be defined as the set of edges* E *of RAG* γ*.*

Since the set V has two disjoint subsets P and Q (by Definition 4), a process can only request a resource (not another process), and similarly a resource can only be granted to a process (but not to another resource); thus, any RAG γ is a bipartite graph, a graph whose vertices can be partitioned into two groups $P \cup Q$ ($P \cap Q = \phi$), where all edges cross from one group to the other group.

[1]Please see Chapter 2 pages 39-40 of [9] for basic definitions from graph theory.

Definition 10. *We denote $\gamma_i = \{V, E\}$ as a particular system. We also define $\gamma_{i1}, \gamma_{i2}, \gamma_{i3}, \ldots$ to be different instances or **states** of the given system γ_i (same set V). Note that the edge set $E(\gamma_{ij})$ is different for each $j \in \{1, 2, 3, \ldots\}$, but the node set $V = \{P \cup Q\}$ is constant for a given system γ_i. The system γ_i changes from one state γ_{ij} to another state γ_{ik} when handling requests, grants and releases of resources.*

Example 2. RAG
Figure 3 (a) shows a RAG in state γ_{ij} converted to state γ_{ik} shown in (b) when a pending request (p_3, q_2) is granted. Here state γ_{ik} consists of a set of processes, $P = \{p_1, p_2, \ldots, p_6\}$, a set of resources, $Q = \{q_1, q_2, \ldots, q_6\}$, a set of request edges $R = \{(p_1, q_4), (p_3, q_1), (p_4, q_6), (p_5, q_4), (p_6, q_5)\}$ or $\{r_{1\to 4},\ r_{3\to 1},\ r_{4\to 6},\ r_{5\to 4},\ r_{6\to 5}\}$, and a set of grant edges $G = \{(q_1, p_5), (q_2, p_3), (q_3, p_5), (q_4, p_3), (q_5, p_1), (q_6, p_2)\}$ or $\{g_{1\rightharpoonup 5},\ g_{2\rightharpoonup 3},\ g_{3\rightharpoonup 5},\ g_{4\rightharpoonup 3},\ g_{5\rightharpoonup 1},\ g_{6\rightharpoonup 2}\}$. $E(\gamma_{ik})$, defined in Definition 9, gives $\{R \cup G\}$.

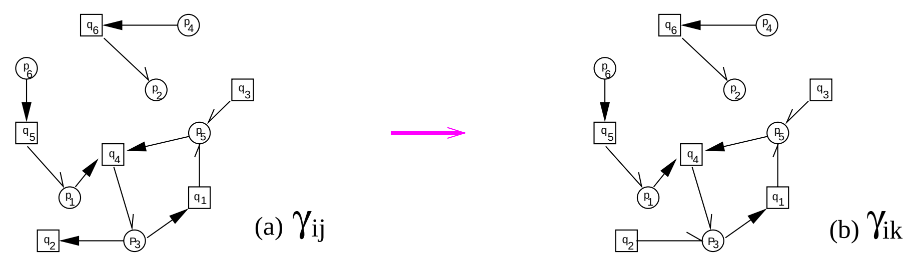


FIGURE 3. RAG example

□

4.2. Definition of a Cycle

Both this subsection and the next subsection further refine various relationships among nodes and give the definition of a cycle. The definitions in this subsection become the basis of the deadlock theorems to be explained in Section 6.

Definition 11. *A node v_τ is a **terminal node** iff the node v_τ has at least one edge and only incoming edge(s) or only outgoing edge(s). (A node with no edges is **not** a terminal node.)*

Definition 12. *An edge connected to a terminal node v_τ is called a* **terminal edge** *e_τ.*

Definition 13. *Given a RAG in state γ_{ij}, let $\tau(\gamma_{ij})$ be a function that returns the set $\{e_{\tau_1}, e_{\tau_2}, \ldots, e_{\tau_p}\}$ of all terminal edges in γ_{ij}.*

Definition 14. *A **link** node v_λ is a node that has exactly one incoming edge and exactly one outgoing edge. Clearly, the number of edges of a link node is two.*

Definition 15. *A **branch** node v_ω has one or more incoming edges and one or more outgoing edges such that the total number of edges is greater than or equal to three.*

Note that while a resource node may have multiple incoming edges (multiple requests for the resource), it may have only one outgoing edge (the resource may be granted only to one process). A process node, in contrast, may have multiple incoming and outgoing edges.

Definition 16. *A node v_ϕ is a **connect** node if and only if node v_ϕ is either a link node or a branch node.*

Definition 17. *A **path** $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$, $k \geq 2$ is a set of nodes connected by a consecutive ordered sequence of alternating request and grant edges (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k), where every node in the path is distinct and where every other node belongs to the same set P or Q.*

Note that a path may have just one edge as well as many edges.

Definition 18. *A **simple path** is a path $(v_i, v_{i+1}, \ldots, v_k)$ such that both (i) v_i and v_k are terminal nodes and (ii) all other nodes v_j, $i < j < k$, are link nodes.*

Definition 19. *A **dangling path** is a path $(v_i, v_{i+1}, \ldots, v_j)$ such that either (i) v_i is a terminal node and v_j is a branch node or (ii) v_i is a branch node and v_j is a terminal node.*

Definition 20. *A **cycle** is a set of nodes $(v_i, v_{i+1}, \ldots, v_j, v_i)$ consisting of a path $(v_i, v_{i+1}, \ldots, v_j)$ and an additional edge between v_j and v_i.*

Note that any subset $(v_k, v_{k+1}, \ldots, v_l) \subset$ cycle C with $|(v_k, v_{k+1}, \ldots, v_l)| \geq 2$ is a path. Note also that all nodes in a cycle are connect nodes.

Example 3. Cycle
The RAG in Figure 4 contains terminal nodes and edges; link, branch and connect nodes; simple and dangling paths; and a cycle. $\{p_2, p_4, p_6, q_2, q_3\}$ are terminal nodes with corresponding terminal edges $\{(q_6, p_2), (p_4, q_6), (p_6, q_5), (q_2, p_3), (q_3, p_5)\}$, respectively. Link nodes are $\{q_6, q_5, p_1, q_1\}$, branch nodes are $\{q_4, p_3, p_5\}$, and connect nodes are all the link and branch nodes, i.e., $\{q_6, q_5, p_1, q_1, q_4, p_3, p_5\}$. An example of a simple path is path $p4 \rightarrow q6 \rightharpoonup p2$. An example of a dangling path is path $p6 \rightarrow q5 \rightharpoonup p1 \rightarrow q4$. Finally, $p3 \rightarrow q1 \rightharpoonup p5 \rightarrow q4 \rightharpoonup p3$ forms a cycle. Please note that while the formal notation for the cycle is $(p_3, q_1, p_5, q_4, p_3)$, we also use $p3 \rightarrow q1 \rightharpoonup p5 \rightarrow q4 \rightharpoonup p3$ as an informal notation for the same cycle.

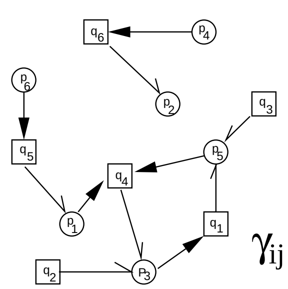


FIGURE 4. A cycle example

□

Definition 21. *The* ***size of a cycle*** *is the number of nodes (both resources and processes) involved in the cycle.*

4.3. Definition of a Terminal Reduction Step

Having introduced a RAG and a cycle, we now define a terminal reduction step and its related properties used to reveal a deadlock quickly and efficiently.

Definition 22. *A* ***terminal reduction step*** *is a step in which at least one terminal edge e_τ is removed from the RAG under consideration.*

Definition 23. *The application of a terminal reduction step to γ_{ij}, resulting in a distinct state $\gamma_{i,j+1}$, is called the* ***reduction*** *of γ_{ij} to $\gamma_{i,j+1}$. Furthermore, we also say that state γ_{ij} has been* ***reduced*** *to $\gamma_{i,j+1}$.*

Definition 24. *If a system state γ_{ij} can be transformed by a terminal reduction step to another state $\gamma_{i,j+1}$, resulting in $\gamma_{i,j+1} \neq \gamma_{ij}$, then the system state γ_{ij} is said to be* ***reducible****. If a system state γ_{ij} cannot be reduced to another different state $\gamma_{i,j+1}$ (because there are no terminal edges to which to apply a terminal reduction step), then system state γ_{ij} is said to be* ***irreducible****.*

Definition 25. *A system state $\gamma_{i,j+k}$ is said to be* ***completely reduced*** *if $E(\gamma_{i,j+k}) = \emptyset$. Otherwise, a system state $\gamma_{i,j+k}$ is said to be* ***incompletely reduced*** *if $E(\gamma_{i,j+k}) \neq \emptyset$.*

5. System Model and Assumptions

Section 2 introduced a future handheld device, shown in Figure 2. Here, we are going to discuss the same issues we discussed in Section 2, but in a more formal way using definitions from Section 4. The following example represents a future SoC. We denote this kind of request-grant system with many resources and processes shown in Figure 5 as γ_i. A system state γ_{ij} of γ_i may have multiple request and grant edges at the same time as shown in Figure 5 (b).

Example 4. A Request-Grant SoC with Multiple PEs and Multiple Resources

We here consider the device shown in Figure 5 as a particular SoC example with multiple PEs and resources. This SoC consists of four PEs and four resources: a video and image capturing interface (VI), an MPEG encoder/decoder, a DSP and a wireless interface (WI), which we refer to as q_1, q_2, q_3 and q_4, respectively, shown in Figure 5 (b). The SoC also contains memory, a memory controller and a deadlock detection unit (DDU) in reconfigurable logic. We denote this SoC as γ_i, and a particular state with a set of requests and grants of the SoC as γ_{ij}. For the sake of simplicity, we assume that currently each PE has only one active process; i.e., each process p_1, p_2, p_3 and p_4 shown in Figure 5 (b) runs on PE1, PE2, PE3 and PE4, respectively. In the current state γ_{ij}, resource q_1 is granted to process p_1, which in turn requests q_2. In the meantime, q_2 is granted to p_3, which requests q_4 while q_4 is granted to process p_4.

In Figure 5, this particular request-grant system γ_i has four processes; thus, $P = \{p_1, p_2, p_3, p_4\}$. γ_i also has four resources, $Q = \{q_1, q_2, q_3, q_4\}$. The DDU in Figure 5 monitors all requests and grants on the bus and decides whether the system has a deadlock or not.

□

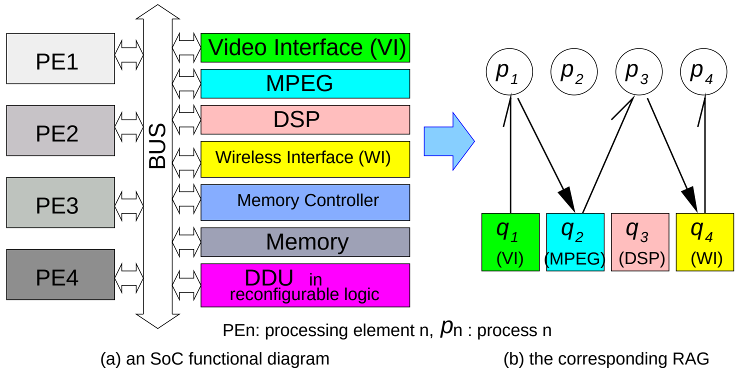


FIGURE 5. A practical SoC realization

Considering this kind of future SoC shown in Figure 5 as our system model, we now introduce some of our assumptions about future SoC designs related to analyzing deadlock in such SoCs.

Assumption 1. *In our system model, only reusable resources exist.*

A **reusable resource** is characterized as follows: (i) Units are neither created nor destroyed (fixed total inventory), and (ii) units are requested and acquired by processes from a pool of available units. When the process finishes using the acquired reusable resource, the resource is returned to the resource pool so that other processes can have a chance to use the resource. Therefore, all further references to "resource" should be read as "reusable resource."

Assumption 2. *Each resource has one unit. Furthermore, each resource can serve only one process at any given time. As a result, a process must wait for all requested unavailable resources to become available before proceeding.*

One consequence of Assumption 2 is that pipelined or multiple-unit resources are not supported. We will show later in Section 6.5 an extension to our approach which allows the relaxing of Assumption 2 and thus the use of pipelined and multiple-unit resources.

Assumption 3. *A resource can only be released by the process holding it.*

Note that in some situations, Assumption 3 may not hold. For example, if the system is reset, then all resources will be released, and all processes will be restarted. However, our analysis is intended to address the normal operation of an SoC. During normal operation, we assume that once a resource is granted to a process, only the owner process can release the resource. In other words, we currently assume that resource preemption is not implemented in the system. In the case that system reset occurs, any deadlock detection operation in progress will have to be restarted with the new system state after reset.

Assumption 4. *While a process holds some resources, the process can request additional resources.*

Note that if an SoC does not satisfy Assumptions 1-4, then the system either (i) cannot exploit the DDU, (ii) must be modified to fit Assumptions 1-4, or (iii) does not need to use the DDU since the SoC will never face deadlock.

6. Methodology

In this section, we will first show how the Deadlock Detection Unit (DDU), the hardware implementation of PDDA, can be used in a multi-processor, multi-resource SoC such as the SoC shown in Figure 5. After that, we will introduce and prove our deadlock theorems and direct consequences confined by our assumptions described in Section 5 (i.e., our deadlock theorems are modified from general deadlock theorems [7]). Then, we will present a translation of a system state γ_{ij} from a RAG into a matrix. This matrix representation forms the basis of PDDA and enables the implementation of simple but very fast parallel deadlock detection in hardware.

Using the matrix representation, we will next define terminal rows and terminal columns to which a novel parallel terminal reduction step (the core of PDDA) can be applied. After that, we will describe PDDA and finally prove that the DDU has a run-time complexity of $O(min(m, n))$, where m and n are the numbers of resources and processes, respectively, involved in deadlock detection.

6.1. Deadlock Detection Unit Operation in a System

First let us briefly explain how the DDU operates and how it can be used in a system. The DDU idles when there are no requests or grants. That is to say, the DDU becomes active and starts working only when a request-, grant- or release-event occurs. Once the DDU is activated, it operates in only a few clock cycles (at most $2 \times min(m, n) - 3$ cycles) and then produces a deadlock detection result. After that, the DDU returns to an idle state and remains idle until another event occurs.

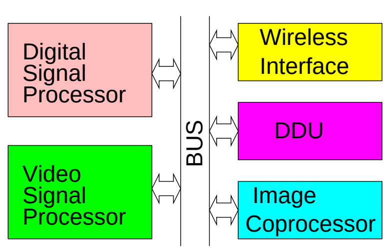
Note that with PDDA implemented in the DDU hardware, the DDU decides whether a given system state has a deadlock or not based on the requests and grants that have occurred, not on any future events.

The DDU can be employed in two ways in a system. One way is that while processes request resources randomly and directly from the resources (i.e., without the intervention of the DDU), the DDU just monitors these requests and grants. As soon as a deadlock is detected, however, the DDU notifies the RTOS of the existence of a deadlock, in which case the intervention is desirable, so the RTOS may abort some processes or take other actions to break the deadlock. Since the deadlock detection takes only a few clock cycles, any further requests and grants during this short amount of time can be temporarily queued in hardware; then, after the current detection, the next detection will start with the update of the queued events. In this way of DDU usage, the DDU does not impede normal system performance at all.

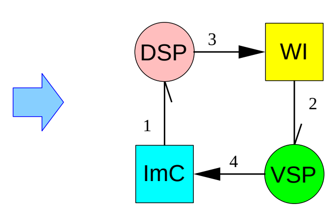
The other way to use the DDU is more conservative than the first way. In this second way, all processes request their resources from the DDU so that the DDU determines if a request or a grant leads to a deadlock and allows the request or the grant only if the action does not lead to a deadlock. In other words, the DDU handles resource allocation and thus can control or limit deadlock situations more aggressively. Consequently, this second way of usage may incur a performance penalty. However, the penalty can be quite small with the DDU since the DDU requires only a few clock cycles for a deadlock check. While detecting deadlock, the DDU indicates a busy signal, indicating that the DDU is currently working; thus temporarily preventing any further events.

In addition to the two ways of DDU usage described here, there may be many more ways to use the DDU. However, the focus on this paper is to prove the correctness and run-time complexity of the DDU. The following example represents the first DDU usage.

Example 5. DDU usage
Consider the SoC shown in Figure 6, which is similar to the one introduced in Section 1. To complete its processing of streaming data, each processor (DSP or VSP) has to use both resources, the WI and ImC, exclusively. In this SoC, a DDU monitors resource request and grant activities. In the case shown in Figure 6, we assume both the DSP and the VSP received a stream to be processed almost at the same time. While the DSP first requests and holds resource ImC, the VSP requests and holds resource WI. (Please see the event sequence marked on the side of each edge shown in Figure 6 (b).) In the meantime, the DDU executes PDDA, which checks for deadlock whenever a request or grant event occurs, but the DDU fails to find a deadlock so far. After that, the DSP requests and waits for the WI, which has already been granted to the VSP. Next, the VSP requests the ImC, and then the DDU starts to determine a deadlock. Since the system state has a deadlock at this instant, the DDU will find the deadlock; thus indicate that a deadlock exists.



(a) an SoC



(b) a RAG

FIGURE 6. A DDU usage example

□

6.2. Deadlock Detection Theorems

Before introducing and proving our deadlock theorems, we now describe one more definition and two underlying assumptions as they relate to the DDU, its operation in practical situations and then our deadlock theorems.

Definition 26. *A process is* ***making progress*** *in a system γ_i in state γ_{ij} only when one of the following is true: (i) when a process does not hold resources, the process, if given control of a processor, could currently perform its computation without the need of any resources, or (ii) when a process holds some resources, the process, if given control of a processor, would currently use the resources to perform its computation without a need for any additional resources.*

Assumption 5. *In a practical situation, at the instant when the DDU becomes active and is checking for deadlock, the DDU determines deadlock based on the requests and grants currently in existence at that instant,* ***not on any future events*** *(i.e., assuming no additional requests or grants are accepted when the DDU is in operation, which can easily be implemented by indicating a "busy" signal).*

Assumption 6. *At the instant when the DDU becomes active (at a particular state of a system), if a process is making progress and using some resources, then it is assumed that the process has all the resources it requires and that it can and will finish using its resources within a finite time and eventually release all the resources that the process has used.*

We now introduce and prove our deadlock theorems and their direct consequences.

Corollary 1. *In a system state γ_{ij}, the number of nodes involved in the smallest possible cycle is four. Similarly, the number of edges involved in the smallest possible cycle is four.*

Proof. Since a RAG γ is a bipartite graph (by Definitions 4 and 8), γ_{ij} cannot have any edge from process p_g to process p_h for any two processes $p_g, p_h \in V(\gamma_{ij})$. Similarly, γ_{ij} cannot have any edge from resource q_k to resource q_l for any two resources $q_k, q_l \in V(\gamma_{ij})$. Since we need to find the minimum number of nodes that can form a cycle, let us consider case (i) where one process and one resource exist. Case (i) can have a path between the two nodes but cannot form a cycle because according to Assumption 2 there cannot exist two edges (i.e., a request and a grant edge) between the two nodes (one process and one resource) that could form a cycle. Now consider case (ii) where two processes and one resource exist. Case (ii) can have a path among them, but they cannot form a cycle because according to the bipartite property of a RAG there cannot exist an edge between the two processes. Similarly, case (iii) where one process and two resources exist cannot form a cycle because there cannot exist an edge between the two resources. Thus, forming a cycle must require at least two distinct processes and two distinct resources. Therefore, forming the smallest cycle requires at least four nodes. Furthermore, forming a cycle with four nodes requires at least four edges since otherwise all four nodes cannot be connect nodes. Therefore, the number of edges involved in the smallest cycle is also four. □

Corollary 2. *In a system state γ_{ij}, the number of edges in any path using all nodes in the smallest possible cycle is three.*

Proof. According to Corollary 1, the smallest possible cycle has four nodes, i.e., two distinct processes and two distinct resources. Let the two processes be p_1, p_2 and the two resources be q_1, q_2. According to the bipartite property of a RAG, there cannot exist an edge between the two processes or between the two resources. Therefore, one longest path is $\{p_1, q_1, p_2, q_2\}$ since this path uses all nodes in γ_{ij}. This path has three edges. There are three more cases of the longest path, and the number of edges in all three cases is also three. That is, the number of edges in any longest path in the smallest possible cycle is three. □

Theorem 1. *If a system γ_i in state γ_{ij} contains a cycle C, then no nodes in cycle C can be excluded from further consideration through any sequence of terminal reduction steps (Definition 22). As a result, cycle C **cannot** be removed through any sequence of terminal reduction steps. That is, the system state γ_{ij} **cannot** be completely reduced (Definition 25).*

Proof. A node can be excluded from further consideration by a terminal reduction step only if after the terminal reduction step, the node does not have any edges. However, every node in cycle C is a connect node and thus must have an incoming edge from another node in cycle C and an outgoing edge to another node in cycle C. That is, none of the edges in cycle C are terminal edges since they are all exclusively connected to connect nodes. Now, according to Definition 22, a terminal reduction step only removes terminal edges; thus, since none of the edges in cycle C are terminal edges, no edge in cycle C can be removed by the first terminal reduction step. Since no edges in cycle C are removed by the first terminal reduction step, all nodes in cycle C remain connect nodes. Thus, for the second terminal reduction step in any sequence, each edge in cycle C remains connected to connect nodes on both ends of the edge. Continuing in this way, we conclude that no edge in cycle C can be removed by any sequence of terminal reduction steps. Hence, since no edge in cycle C can be removed, no node in cycle C can be excluded from further consideration. Furthermore, since neither nodes nor edges in cycle C can be removed, cycle C itself cannot be removed. Therefore, according to Definition 25, γ_{ij} cannot be completely reduced. $\square$

Lemma 1. *Given system γ_i in state γ_{ij} with cycle C, removing terminal edges (i.e., edges connected to terminal nodes) will not alter cycle C.*

Proof. Every node in cycle C is a connect node. Furthermore, every node in cycle C must have an edge **to** another node in cycle C and **from** another node in cycle C. Therefore, if a node in cycle C has an edge to or from a terminal node, the terminal node cannot be in cycle C. Thus, the removal of an edge to or from a terminal node leaves cycle C intact since none of the edges from a node in cycle C to other nodes in cycle C are edges to or from terminal nodes. $\square$

Consider a sequence of terminal reductions steps (Definition 22) applied to a given γ_{ij}, resulting in an irreducible system state $\gamma_{i,j+k}$. According to the definition of **irreducible** (Definition 24), $\gamma_{i,j+k}$ has no **terminal** edges, resulting in two cases: (i) $\gamma_{i,j+k}$ is completely reduced or (ii) $\gamma_{i,j+k}$ is incompletely reduced (Definition 25). We will next prove that in case (i) γ_{ij} does not have a deadlock while in case (ii) γ_{ij} has a deadlock.

Lemma 2. *If a system state γ_{ij} can be completely reduced, then it does not have a deadlock.*

Proof. A complete reduction deletes all edges including all request edges. Since a request edge can be deleted only if the request could be fulfilled within a finite time, deleting all the request edges implies that all processes can eventually obtain the resources that they have requested. As a result, all the processes can make progress (Definition 26). This fact violates the deadlock definition (Definition 1); hence, γ_{ij} does not have a deadlock.[2] $\square$

Theorem 2. *If a system state γ_{ij}* ***cannot*** *be completely reduced, then the system contains at least one cycle.*

Proof. If γ_{ij} cannot be completely reduced, then a sequence of reduction steps applied to γ_{ij} results in irreducible state $\gamma_{i,j+k}$ with the property $E(\gamma_{i,j+k}) \neq \emptyset$. In other words, $\gamma_{i,j+k}$ is irreducible and has some edges.

We next note that all the nodes connected to edges in $\gamma_{i,j+k}$ must be connect nodes (since $\gamma_{i,j+k}$ does not contain any terminal nodes). Consider an arbitrary connect node v_ϕ in $\gamma_{i,j+k}$.

[2] Lemma 2 is equivalent to Corollary 1 on page 189 in [4].

Taking v_ϕ's outgoing edge (there must be at least one outgoing edge from v_ϕ since v_ϕ is a connect node) we arrive at node $v_{\phi+1}$. Note that $v_{\phi+1} \neq v_\phi$ since our system does not have any edges from a node back to the same node. Note also that the edge we took to arrive at $v_{\phi+1}$ is an edge incoming to $v_{\phi+1}$. However, since $v_{\phi+1}$ must also be a connect node, an edge must be outgoing from $v_{\phi+1}$. Taking this outgoing edge, we arrive at another node. Continuing in this way, every node must be connected to another node distinct from itself. Eventually, we arrive either (i) at a node $v_{\phi+m}$ that was previously visited already, or else (ii) at the last node in the graph. In case (i), we have a cycle. In case (ii), this last node in the graph must be a connect node. Since all the nodes have already been visited, the outgoing edge of this last node must lead to a node already previously visited. Thus, in case (ii) we have a cycle as well. Therefore, if a system state γ_{ij} is not completely reducible, then the system contains at least one cycle. □

Lemma 3. *If no cycle exists in a system state γ_{ij}, then γ_{ij} can be completely reduced.*

Proof. This lemma is the contraposition of Theorem 2, and it is well-known that the contraposition of a proposition is always true provided that the given proposition is true. That is, if a system state γ_{ij} cannot be completely reduced, then γ_{ij} contains at least one cycle, which implies that if no cycle exists in γ_{ij}, then the system state γ_{ij} is completely reducible. □

Lemma 4. *In a system γ_i in state γ_{ij}, a process p_k that is making progress cannot be involved in deadlock.*

Proof. Given a system γ_i in state γ_{ij}, if a process p_k is making progress, then according to Definition 26, two cases result: (i) p_k does not need any resources, or (ii) p_k has some resources. In case (i), if p_k is making progress, and it does not need any resources, then according to the definition of deadlock (Definition 1), p_k has nothing to do with deadlock. In case (ii), if p_k that has some resources is making progress, then, according to Assumption 6, p_k has all required resources, it will finish using the resources, and it will then release the resources; thus, p_k has, at this instant, no unfulfilled resource requests preventing its progress (and eventual release of the resources it does hold). Therefore, p_k does not fulfill Condition 4 (see Section 1); thus cannot be involved in deadlock. As a result, in both cases (i) and (ii), p_k, which is making progress, is not involved in deadlock. □

Lemma 5. *If system state γ_{ij} in system γ_i does* ***not*** *have a deadlock, then all processes in the system state can* ***make progress*** *either now or at some time a finite distance in the future.*

Proof. According to the definition of deadlock (Definition 1), even if one process is blocked while waiting for requirements that can never be satisfied, the system state has a deadlock. Therefore, lest γ_{ij} have a deadlock, no process exists that is unable to make progress within a finite time. In other words, if γ_{ij} does not have a deadlock, all processes in γ_{ij} must be able to make progress within a finite time. □

Note that when a process in a system γ_i in state γ_{ij} acquires a resource for which the process has waited, the corresponding request edge is removed and changed to a grant edge. Accordingly, as requests are fulfilled, request edges are replaced with grant edges. After using the granted resources, a process will eventually release the resources that it has used. Since releasing a resource is expressed as removing a grant edge, a process that releases all of its resources will lose all of its grant edges.

Theorem 3. *Given system γ_i in state γ_{ij} and under Assumptions 1-6, a* ***cycle*** *is a necessary and sufficient condition for deadlock.*

Proof. We prove this theorem by contradiction. Suppose that γ_{ij} has a cycle but does not have a deadlock. If γ_{ij} does not have a deadlock, then from Lemma 5 all processes in γ_{ij} can make progress either now or at some time a finite distance in the future. According to Definition 26 and Assumption 6, a process that requires some resources can only make progress when the process obtains all the resources for which the process is waiting. Thus, all the processes being able to make progress within a finite time implies that all processes will (eventually) obtain all needed resources, finish using them, and release them within a finite time. In other words, considering the RAG representation, as processes receive resources, request edges will be changed into corresponding grant edges, and then all the grant edges will eventually be removed as processes release resources. Thus, after all computations are finished, there will exist no edges in the final state $\gamma_{i,j+k}$. Note that, as stated in Assumption 5, we are considering the case in which no new requests come in during the completion of all computations implied by state γ_{ij}. Since there are no edges after all computations are finished, and since by Theorem 1 a cycle that once exists cannot disappear through terminal reduction steps, there cannot exist a cycle in γ_{ij} at all. This contradicts our supposition that γ_{ij} has a cycle, but it does not have a deadlock. Therefore, if γ_{ij} has a cycle, then γ_{ij} has a deadlock.

Now assume that γ_{ij} has no cycle but has a deadlock. If γ_{ij} has no cycle, then according to Lemma 3, γ_{ij} is completely reducible. This complete reduction indicates no deadlock according to Lemma 2. Therefore, according to Lemmas 2 and 3, γ_{ij}, which has no cycle, cannot have any deadlock, which, however, contradicts our assumption that γ_{ij} has a deadlock. Thus, if γ_{ij} has a deadlock, then at least one cycle must exist in γ_{ij}. As a result, a **cycle** is a necessary and sufficient condition for deadlock. □

Although it has already been proven in [2] and [4] that a cycle is a necessary and sufficient condition for deadlock, none of the proofs were exactly applicable to our system model with a DDU. Therefore, rather than adapting our system model and notation to prior proofs, we decided to prove Theorem 4 in the exact context that we wanted for our claims of correctness for the proposed PDDA and its hardware implementation in the DDU.

6.3. PDDA Definitions

So far, we have covered terms and properties that are applied to PDDA. As we proposed, our deadlock detection approach dramatically reduces deadlock detection time by implementing PDDA in hardware. In order to easily accommodate such hardware, we propose that the RAG corresponding to state γ_{ij} be mapped into a matrix M_{ij} that will have exactly the same request and grant edges as the RAG corresponding to γ_{ij} but with another notation for each edge. We now define a RAG matrix and a terminal reduction step before introducing an algorithm that exploits these.

Definition 27. *The purpose of this definition is to define matrices that correspond to graph γ, system γ_i and state γ_{ij} from Definitions 8 and 10. A **RAG matrix M** is a matrix mapped from a RAG γ and represents an arbitrary system with processes and resources. A **system matrix M_i** is defined as a matrix representation of a particular system γ_i, where the rows (fixed in size) of matrix M_i represent the fixed set Q of resource nodes of γ_i, and the columns (fixed in size) of matrix M_i represent the fixed set P of process nodes of γ_i. We denote another notation of this relationship as $M_i \equiv \gamma_i$ for the sake of simplicity. A **state matrix M_{ij}** is a matrix that represents a particular system state γ_{ij}, i.e., $M_{ij} \equiv \gamma_{ij}$. Edges in system state γ_{ij} are mapped into the corresponding array elements using the following rule:*

Given $E = \{R \cup G\}$ from γ_{ij},

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

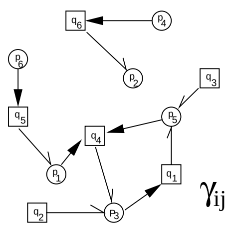
for all rows $1 \le s \le m$ *and for all columns* $1 \le t \le n$*:*

$\alpha_{st} = g_{s \rightharpoonup t}$ *(or simply 'g'), if there exists a grant edge* $(q_s, p_t) \in G$

$\alpha_{st} = r_{t \to s}$ *(or simply 'r'), if there exists a request edge* $(p_t, q_s) \in R$

$\alpha_{st} = 0_{st}$ *('0' or a blank space), otherwise.*

Example 6. State Matrix Representation
The system in state γ_{ij} shown in Figure 7 (a) can be represented in the matrix form shown in (b). For the sake of better understanding, we will use the matrix representation shown in Figure 7 (c) from now on.



(a)

$$\boldsymbol{M_{ij}} = \begin{bmatrix} 0 & 0 & r & 0 & g & 0 \\ 0 & 0 & g & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g & 0 \\ r & 0 & g & 0 & r & 0 \\ g & 0 & 0 & 0 & 0 & r \\ 0 & g & 0 & r & 0 & 0 \end{bmatrix} \Longrightarrow$$

(b)

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6		g		r		

(c)

FIGURE 7. Matrix representation example

□

Definition 28. *A state matrix* M_{ij} *is said to be* **reducible** *if its corresponding system state* γ_{ij} *is reducible (Definition 24). Similarly,* M_{ij} *is said to be* **irreducible** *if its corresponding* γ_{ij} *is irreducible.*

Definition 29. *A system matrix* $M_{i,j+k}$ *is said to be* ***completely reduced*** *if its corresponding system state* $\gamma_{i,j+k}$ *is completely reduced (Definition 25). Likewise,* $M_{i,j+k}$ *is said to be* **incompletely reduced** *if its corresponding* $\gamma_{i,j+k}$ *is incompletely reduced.*

Definition 30. *A* ***terminal row*** $\boldsymbol{\tau_s}$ *is a row s (recall that row s corresponds to resource* q_s*) of matrix* M_{ij} *such that either (i) all non-zero entries* $\{\alpha_{st_r} \neq 0,\ 1 \le t_r \le n\}$ *are request entries* $r_{t_r \to s}$ *with at least one request entry (i.e., one or more request entries and no grant entries in the row) or (ii) one entry* α_{st_g}*,* $1 \le t_g \le n$*, is a grant* $g_{s \rightharpoonup t_g}$ *with the rest of the entries* $\{\alpha_{st},$ $1 \le t \le n,\ t \neq t_g\}$ *equal to zero.*

Note that for a terminal row τ_s, its corresponding resource node q_s is a terminal node; thus, all non-zero entries in row τ_s are terminal edges. Note also the effect of Assumption 2 stated in Section 5 that a resource can only be granted to one process. Specifically, the effect is that in case (ii) in Definition 30, a row (corresponding to a resource) may have at most one grant entry in the row.

Example 7. Terminal Row

In Figure 8, row q_2 is a terminal row according to case (i) of Definition 30. Also, row q_5 is another terminal row, this time according to case (ii) of Definition 30.

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1						
q_2			r		r	
q_3						
q_4	r		g			
q_5	g					
q_6						

FIGURE 8. Terminal row example

□

Definition 31. *A **terminal column** τ_t is a column t (recall that column t corresponds to process p_t) of matrix M_{ij} such that either (i) all non-zero entries $\{\alpha_{st} \neq 0,\ 1 \leq s \leq m\}$ are request entries with at least one request entry (i.e., one or more request entries and no grant entries in the column) or (ii) all non-zero entries $\{\alpha_{st} \neq 0,\ 1 \leq s \leq m\}$ are grant entries with at least one grant entry (i.e., one or more grant entries and no request entries in the column).*

Note that in a terminal column τ_t, its corresponding process node p_t is a terminal node, and all non-zero entries in the terminal column τ_t are terminal edges.

Example 8. Terminal Column

In Figure 9, column p_5 is a terminal column according to case (i) of Definition 31. Also, column p_3 is another terminal column, this time according to case (ii) of Definition 31.

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1					r	
q_2			g			
q_3						
q_4	r		g		r	
q_5	g					
q_6						

FIGURE 9. Terminal column example

□

Definition 32. *A **terminal reduction step** ϵ is a unary operator $\epsilon : M_{ij} \mapsto M_{i,j+1}$, where ϵ calculates the terminal edge set $\tau(M_{ij}) = \tau(\gamma_{ij})$ defined in Definition 13 and returns $M_{i,j+1}$ such that all terminal edges $\tau(M_{ij})$ found are removed by setting the terminal entries found to zero; thus, the next iteration $M_{i,j+1}$ will start with equal or fewer total edges as compared to M_{ij}. This terminal reduction step is denoted as $\epsilon(M_{ij})$. The formula for $M_{i,j+1} = \epsilon(M_{ij})$ is shown in Equation 1 (see Definition 27 for the meaning of $M_{i,j+1} \equiv \gamma_{i,j+1}$):*

$$
(1)\quad \begin{aligned} M_{i,j+1} &= \epsilon(M_{ij}) && \equiv \epsilon(\gamma_{ij}) \\ &= \{V, E(M_{ij}) - \tau(M_{ij})\} && \equiv \{V, E(\gamma_{ij}) - \tau(\gamma_{ij})\} = \gamma_{i,j+1} \end{aligned}
$$

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

Example 9. One Step of Terminal Reduction (ϵ)
Figure 10 (b) shows a new matrix $M_{i,j+1}$ after the matrix reduction step $\boldsymbol{\epsilon}$ defined in Definition 32 is applied to M_{ij} shown in (a). In matrix M_{ij}, q_2 and q_3 are terminal nodes by Definition 11 (also terminal rows by Definition 30); thus, all the edges in their rows are terminal edges by Definition 12. Therefore, all the edges in rows q_2 and q_3 can be removed. Likewise, p_2, p_4 and p_6 are terminal nodes (also terminal columns by Definition 31); hence, all edges in these columns can be removed, resulting in matrix $M_{i,j+1}$.

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6		g		r		

(a)

$\overset{\boldsymbol{\epsilon}}{\Longrightarrow}$

$M_{i,j+1}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4	r		g		r	
q_5	g					
q_6						

(b)

FIGURE 10. A terminal reduction step (ϵ) example

□

Definition 33. ***A terminal reduction sequence $\boldsymbol{\xi}$***, *applicable to a matrix M_{ij}, is a sequence of k terminal reduction steps ϵ (recall that ϵ is a terminal reduction step) such that (i) $M_{ij} \mapsto M_{i,j+1} \mapsto \cdots \mapsto M_{i,j+k}$; (ii) $M_{i,j+k}$ is irreducible; and (iii) $\{M_{i,j+h}, 0 \leq h < k\}$ are all reducible. A terminal reduction sequence is called a* ***complete reduction*** *when the sequence of terminal reduction steps corresponding to ξ results in $M_{i,j+k}$ such that the irreducible state matrix $M_{i,j+k}$ contains all zero entries (note that this means that $\gamma_{i,j+k}$ corresponding to $M_{i,j+k}$ has no edges: $E(\gamma_{i,j+k}) = \emptyset$). A terminal reduction sequence is called an* ***incomplete reduction*** *when ξ returns $M_{i,j+k}$ with at least one non-zero entry (note that this means that $\gamma_{i,j+k}$ corresponding to $M_{i,j+k}$ has at least one edge: $E(\gamma_{i,j+k}) \neq \emptyset$). Another representation of a terminal reduction sequence is shown in Equation 2.*

$$
\begin{aligned}
M_{i,j+k} &= \xi(M_{ij}) \\
&= \epsilon^{(k)}(\ldots\epsilon^{(2)}(\epsilon^{(1)}(M_{ij}))\ldots) \\
&= \epsilon(\ldots\epsilon(\epsilon(M_{ij}))\ldots)
\end{aligned}
\tag{2}
$$

We here briefly mention the similarities between the concept of terminal reduction and the two-level logic minimization, whose general concept is described in Chapter 2 of [9]. The Quine-McCluskey method used to find minimal prime implicants iteratively reduces "a set of standard sum of products form" to "a set of prime implicants" by using the concepts of dominance and essentiality [24]. By contrast, Coudert et al. proposed an efficient algorithm that directly compute a set of essential elements using binary decision diagrams (BDDs) and metaproducts, which make its complexity independent from the numbers of minterms and prime implicants of a function, without computing the dominance relations, whose huge BDDs are the bottleneck of previous algorithms [8]. Our approach of parallel terminal reduction based on a matrix is very similar to these two-level logic minimization techniques.

6.4. PDDA and Its Proofs

In this section, we first introduce the terminal reduction algorithm which implements $\boldsymbol{\xi}$ (Definition 33). We next show PDDA which uses the terminal reduction algorithm. Finally, we prove the correctness of PDDA and PDDA's run-time complexity when implemented in parallel hardware (i.e., the DDU).

Algorithm 1. *Terminal Reduction Algorithm*

```
ξ(M_ij) {
    k = 0;
    M_iter = M_ij;
    while (1) {
        /* parallel on */
        determine all terminal rows τ_s;
        determine all terminal columns τ_t;
        /* parallel off */
        if (neither ∃τ_s nor ∃τ_t) break; /* if no more terminals */
        /* parallel on */
        for all τ_s, set all entries in rows s of M_iter to zero;
        for all τ_t, set all entries in columns t of M_iter to zero;
        /* parallel off */
        k = k + 1;
    }
    M_i,j+k = M_iter;
    return M_i,j+k;
```

14 }

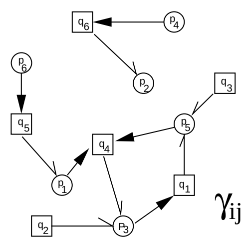
Algorithm 1 is an implementation of the terminal reduction sequence $\boldsymbol{\xi}$ shown in Definition 33. We summarize the operation of Algorithm 1. Lines 2 and 3 of Algorithm 1 initialize two variables: iterator k and matrix M_{iter} that is a copy of input matrix M_{ij}. Line 5 finds all terminal rows (Definition 30), and line 6 finds all terminal columns (Definition 31). Line 7 checks whether M_{iter} is reducible further or not (Definition 28). Lines 8 and 9 remove all the terminal edges found at the current iteration. On the whole, the terminal reduction step $\epsilon(M_{ij})$ of Definition 32 corresponds to lines 5-9 of Algorithm 1, which iterates until the matrix M_{iter} becomes irreducible. Note that, in hardware implementation, lines 5 and 6 of Algorithm 1 are executed at the same time, as are lines 8 and 9.

Algorithm 2. *Parallel Deadlock Detection Algorithm (PDDA)*

1 *Deadlock_Detect_Matrix* (γ_{ij}) {
2 $\quad M[s,t] = [\alpha_{st}]$, *where*
3 $\quad\quad s = 1, \ldots, m$ *and* $t = 1, \ldots, n$
4 $\quad\quad \alpha_{st} = r$, *if* $\exists (p_t, q_s) \in E(\gamma_{ij})$
5 $\quad\quad \alpha_{st} = g$, *if* $\exists (q_s, p_t) \in E(\gamma_{ij})$
6 $\quad\quad \alpha_{st} = 0$, *otherwise.*
7 $\quad M_{i,j+k} = \xi(M_{ij})$; */* call Algorithm 1 */*
8 $\quad$ *if* $(M_{i,j+k} == [\mathbf{0}])$ { */* matrix of all zeros */*
9 $\quad\quad$ *return 0; /* no deadlock */*
10 $\quad$ } *else* {
11 $\quad\quad$ *return 1; /* deadlock detected */*
12 $\quad$ }
13 }

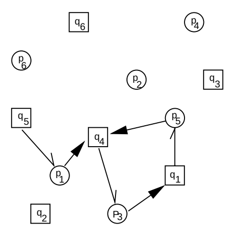
We now summarize the operation of Algorithm 2 (i.e., PDDA). Lines 2-6, given γ_{ij}, construct the corresponding matrix M_{ij} according to Definition 27. Next, line 7 calls Algorithm 1 with argument M_{ij}. When Algorithm 1 is completed, lines 8-12 of Algorithm 2 determine whether γ_{ij} has a deadlock or not by considering returned matrix $M_{i,j+k}$: if $M_{i,j+k}$ is empty, the corresponding γ_{ij} has no deadlock; otherwise, deadlock(s) exist. Finally, Algorithm 2 returns '1' if the system state under consideration has deadlock(s), or '0' if no deadlock. Note that Algorithm 2, which includes Algorithm 1, is referred to as PDDA. Next, we present a simple example that shows the results at each iteration of PDDA.

Example 10. Matrix Reduction Sequence and Deadlock Detection
Let's consider the system γ_i shown in Figure 7 again. Figure 11 illustrates how each step of Algorithm 1 is applied to matrix M_{ij} and its RAG representation γ_{ij}. The original γ_{ij} and M_{ij} are shown at step 0 in Figure 11 (a). After one iteration of lines 5-10 in Algorithm 1, the state becomes step 1 ($M_{i,j+1}$) shown in (b).



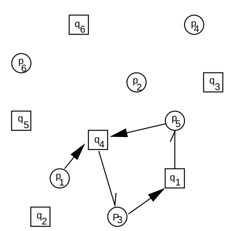
M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2			g			
q_3					g	
q_4	r		g		r	
q_5	g					r
q_6		g		r		

(a) step 0



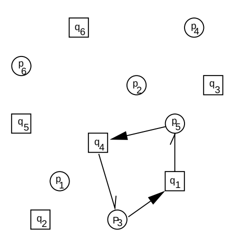
$M_{i,j+1}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4	r		g		r	
q_5	g					
q_6						

(b) step 1



$M_{i,j+2}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4	r		g		r	
q_5						
q_6						

(c) step 2



$M_{i,j+3}$	p_1	p_2	p_3	p_4	p_5	p_6
q_1			r		g	
q_2						
q_3						
q_4			g		r	
q_5						
q_6						

(d) step 3

FIGURE 11. Sample sequence of reduction steps

One more iteration produces step 2 ($M_{i,j+2}$) shown in (c). After the third iteration, Algorithm 1 returns an irreducible matrix $M_{i,j+3}$ shown in (d); thus, Algorithm 2 finds a deadlock by evaluating the returned matrix $M_{i,j+3}$.

□

Theorem 4. *PDDA detects deadlock if and only if there exists a cycle in state γ_{ij}.*

Proof. Consider M_{ij} corresponding to γ_{ij} (Definition 27). (a) Algorithm 1 returns, by construction, an irreducible matrix $M_{i,j+k}$. (b) By the definition of **irreducible** (Definition 28), $M_{i,j+k}$ has no terminal edges, yielding two cases: (i) $M_{i,j+k}$ is completely reduced, or (ii) $M_{i,j+k}$ is incompletely reduced (Definition 29). In case (i), by Lemma 2, M_{ij} (i.e., γ_{ij}) has no deadlock. In case (ii), by Theorem 2, M_{ij} (i.e., γ_{ij}) has at least one cycle; thus, by Theorem 3, γ_{ij} has a deadlock. □

Lemma 6. *In a RAG γ_{ij}, an upper bound on the number of edges in a path is $2 \times min(m,n)$, where m is the number of resources and n is the number of processes.*

Proof. By Definitions 4 and 8, RAG γ_{ij} is a bipartite graph in which any request edge spans from a process node to a resource node and any grant edge spans from a resource node to a process node. That is, γ_{ij} cannot have any edge from process p_g to process p_h for any two processes p_g, $p_h \in V(\gamma_{ij})$. Similarly, γ_{ij} cannot have any edge from resource q_k to resource q_l for any two resources q_k, $q_l \in V(\gamma_{ij})$. Also recall that every node in each path of γ_{ij} is distinct by Definition 17.

Let us consider the following three possibilities: (i) $m = n$, (ii) $m > n$, or (iii) $m < n$. For case (i), where m equals n, one longest path is $\{p_1, q_1, p_2, q_2, \ldots, p_m, q_m\}$ since this path uses all the nodes in γ_{ij}, and since every node in a path must be distinct (i.e., every node can only be listed once). In this case, the number of edges involved in the path is $2 \times m - 1$ (or $2 \times n - 1$). For case (ii), where m is greater than n (i.e., $m - n > 0$), one longest path is $\{q_1, p_1, q_2, p_2, \ldots, q_n, p_n, q_{n+1}\}$; this path cannot be lengthened since every node in a path must be distinct, and since all n process-nodes are already used in the path. Therefore, the number of edges in this path is $2 \times n$. Likewise, for case (iii), where n is greater than m (i.e., $n - m > 0$), the number of edges involved in any longest path is $2 \times m$.

As a result, cases (i), (ii) and (iii) show that the number of edges of the maximum possible longest path in a RAG γ_{ij} is $2 \times min(m,n)$. □

Theorem 5. *Algorithm 1, when implemented in parallel hardware, completes its computation in at most $2 \times min(m,n) - 3 = O(min(m,n))$ steps, where m is the number of resources and n is the number of processes.*

Proof. Given M_{ij}, we consider the corresponding γ_{ij}, which has a one-to-one correspondence with M_{ij} according to Definition 27. At each iteration in line 7 of Algorithm 1, if line 7 evaluates to false, then there exist at least one terminal row or column; thus, another iteration needs to continue. Note that the worst case number of iterations will occur when γ_{ij} has the longest reducible path. However, the simple longest reducible path cannot give the worst case because such a simple path (see Definition 18) has a terminal node at each end of the simple path; thus, each ϵ reduction will remove two edges at a time! Thus, maximizing the number of iterations requires that one end of the longest path be a terminal node while the other end of the longest path connects to a cycle, preventing this end from being reduced.

Now consider the case where γ_{ij} has the longest possible path that is connected to a cycle at one end of the path. According to Lemma 6, since the number of edges of the maximum possible longest path in a bipartite RAG has an upper bound of $2 \times min(m, n)$, the number of iterations in the worst case is bound by $2 \times min(m, n)$. Furthermore, since one end of the path is connected to a cycle, the number of iterations will further be reduced by the path edges contained in the cycle (Definition 20) since edges in a cycle can never be terminal edges. The worst case (i.e., the maximum number of iterations) occurs when the size of the cycle is at its minimum. According to Corollary 2, when all the nodes in the smallest possible cycle are used, the longest path has three edges in this smallest possible cycle. Therefore, in the worst case, $2 \times min(m, n) - 3$ is an upper bound on the number of edges in the longest possible path that are not also part of a cycle.

Hence, the number of iterations required to reach an irreducible state becomes at most $2 \times min(m, n) - 3 = O(min(m, n))$ in the worst case. □

The next example shows terminal reduction steps (Algorithm 1) and a run-time complexity calculation for a simple path case.

Example 11. Run-time Complexity Calculation in a Simple Path Case

M_{ij}	p_1	p_2	p_3	p_4	p_5	p_6
q_1	g	r				
q_2		g	r			
q_3			g	r		
q_4				g	r	
q_5					g	r

FIGURE 12. A simple path example

A RAG for the matrix shown in Figure 12 is

$$p_1 \leftharpoonup q_1 \leftarrow p_2 \leftharpoonup q_2 \leftarrow p_3 \leftharpoonup q_3 \leftarrow p_4 \leftharpoonup q_4 \leftarrow p_5 \leftharpoonup q_5 \leftarrow p_6$$

Due to the two terminal nodes at both ends, after the first iteration of Algorithm 1, the RAG becomes

$$q_1 \leftarrow p_2 \leftharpoonup q_2 \leftarrow p_3 \leftharpoonup q_3 \leftarrow p_4 \leftharpoonup q_4 \leftarrow p_5 \leftharpoonup q_5$$

After the second iteration of Algorithm 1, the RAG becomes

$$p_2 \leftharpoonup q_2 \leftarrow p_3 \leftharpoonup q_3 \leftarrow p_4 \leftharpoonup q_4 \leftarrow p_5$$

After the fourth iteration of Algorithm 1, the RAG becomes

$$p_3 \leftharpoonup q_3 \leftarrow p_4$$

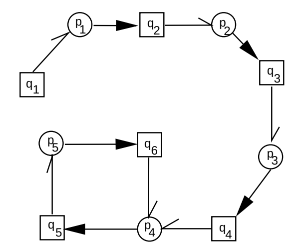
which, after the fifth iteration, leaves no edges.

As shown in this example, in general, the number of iterations required to reach an irreducible state in a simple path case is $min(m, n)$, which, in this case, is 5.

□

The next example illustrates one of the worst cases of terminal reduction.

Example 12. Run-time Complexity Calculation in the Worst Case
In the dangling path shown in Figure 13, it takes 7 iterations to remove all reducible terminal edges. The size of the cycle shown in Figure 13 is four (i.e. the minimum possible). Thus, according to Theorem 5, the number of iterations to reach an irreducible state for the RAG shown in Figure 13 is $(2 \times min(m, n) - 3)$; i.e., $(2 \times 5 - 3) = 7$ iterations.



(a) a RAG

M_{ij}	p_1	p_2	p_3	p_4	p_5
q_1	g				
q_2	r	g			
q_3		r	g		
q_4			r	g	
q_5				r	g
q_6				g	r

(b) M_{ij}

FIGURE 13. A dangling path connected to a cycle when $m \neq n$

□

Examples 11 and 12 demonstrate for specific cases that Algorithm 1 has a time complexity of $2 \times min(m, n) - 3 = O(min(m, n))$ when PDDA is implemented in hardware and executed in parallel.

6.5. Supporting Pipelined Resources and Multiple-Unit Resources

In this subsection, we extend the DDU to support pipelined resources and multiple-unit resource. We can relax Assumption 2 in the following way. First, we associate a unique counter with each pipelined resource or multiple-unit resource. Then, we set the threshold of the counter to the number of pipeline stages or multiple units so that the DDU allows grants until the counter reaches its threshold. If the number of grants is at its threshold, the resource cannot be granted any more.

7. Hardware Implementation of PDDA

In this section, we will present a detailed explanation of PDDA, considering that PDDA is to be implemented in hardware (the DDU). In Section 7.1, we will explain a series of logical operations performed on M_{ij} more generally in terms of matrix theory. Next, in Section 7.2, we will provide two detailed examples to demonstrate each logical operation of PDDA using this matrix theory and then describe the architecture of the DDU in detail. To see the examples before the theoretical descriptions, readers may go directly to Section 7.2.

7.1. Step-by-step Operations of PDDA in hardware (i.e., DDU)

As described in Section 6.1, when the DDU notices an event of either a request, grant or release, a series of logic operations occurs: (i) finding terminal nodes, (ii) reducing terminal edges, (iii) iterating (i) and (ii) until no more terminal edges exist, and (iv) checking for deadlock.

A given system state γ_{ij} is equivalently represented by a system state matrix M_{ij} so that, based on M_{ij}, the DDU can perform the sequence of operations shown in Algorithms 1 and 2 in Section 6.4 and decide whether the given state has a deadlock or not. In the general case, a system state matrix can be explicitly represented as shown in Equation 3.

$$\boldsymbol{M_{ij}} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1t} & \dots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2t} & \dots & \alpha_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha_{s1} & \alpha_{s2} & \dots & \alpha_{st} & \dots & \alpha_{sn} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha_{m1} & \alpha_{m2} & \dots & \alpha_{mt} & \dots & \alpha_{mn} \end{bmatrix} = M_{iter} \text{ (at the first iteration)} \tag{3}$$

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

We now explain the operation of Algorithms 1 and 2 line-by-line in detail. Lines 2-6 of Algorithm 2 illustrate the translation from a given system state γ_{ij} into its matrix form M_{ij}. Each matrix element α_{st} represents one of the following: $g_{s \to t}$ (a grant edge), $r_{t \to s}$ (a request edge) or 0_{st} (no edge). That is, α_{st} is ternary-valued. The hardware implementation of PDDA in digital logic requires that the values of each element be represented in a binary format with a minimum number of encoding bits, preferably. Since α_{st} is ternary-valued, α_{st} can be minimally defined as a pair of two bits $\alpha_{st} = (\alpha^r_{st}, \alpha^g_{st})$. If an entry α_{st} is a grant edge g, bit α^r_{st} is set to 0, and α^g_{st} is set to 1; if an entry α_{st} is a request edge r, bit α^r_{st} is set to 1, and α^g_{st} is set to 0; otherwise, both bits α^r_{st} and α^g_{st} are set to 0. Hence, an entry α_{st} can be only one of the following binary encodings: 01 (a grant edge), 10 (a request edge) or 00 (no activity). This way of element representation is a variant of the positional cube notation developed for logic minimization of digital circuits.[3]

[3] [8] and Chapter 2 of [9]

Next, line 7 of Algorithm 2 calls Algorithm 1. Note that since this process occurs in hardware, an actual call will not occur. Instead of the call in Algorithm 1, the DDU just starts operation with M_{ij}; that is, M_{ij} just becomes M_{iter} during iterations, as shown in line 3 of Algorithm 1. Then, M_{iter} is processed in row-wise and column-wise directions simultaneously, i.e., two-dimensional operation. The row-wise direction operation corresponds to line 5 of Algorithm 1, which finds all terminal rows in M_{iter}. The column-wise direction operation corresponds to line 6 of Algorithm 1, which finds all terminal columns. In order to facilitate the two-dimensional operation, we introduce two more matrices, M^c_{iter} and M^r_{iter}. Even though these two matrices are not actually implemented as separate memory locations in hardware, we nonetheless represent the two-dimensional operation with two-bit binary encoding column and row representations, M^c_{iter} and M^r_{iter} (which are shown in Equation 4 and Equation 5, respectively) so that understanding parallel two-dimensional operation inside the DDU can become easier.

$$\boldsymbol{M^c_{iter}} = \begin{bmatrix} (\alpha^r_{11}, \alpha^g_{11}) & \dots & (\alpha^r_{1t}, \alpha^g_{1t}) & \dots & (\alpha^r_{1n}, \alpha^g_{1n}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^r_{s1}, \alpha^g_{s1}) & \dots & (\alpha^r_{st}, \alpha^g_{st}) & \dots & (\alpha^r_{sn}, \alpha^g_{sn}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^r_{m1}, \alpha^g_{m1}) & \dots & (\alpha^r_{mt}, \alpha^g_{mt}) & \dots & (\alpha^r_{mn}, \alpha^g_{mn}) \end{bmatrix} \tag{4}$$

$$\boldsymbol{M^r_{iter}} = \left[\begin{array}{ccccc} \alpha^r_{11} & \dots & \alpha^r_{1t}, & \dots & \alpha^r_{1n} \\ \alpha^g_{11} & \dots & \alpha^g_{1t} & \dots & \alpha^g_{1n} \\ \hline \vdots & \vdots & \vdots & \vdots & \vdots \\ \hline \alpha^r_{s1} & \dots & \alpha^r_{st} & \dots & \alpha^r_{sn} \\ \alpha^g_{s1} & \dots & \alpha^g_{st} & \dots & \alpha^g_{sn} \\ \hline \vdots & \vdots & \vdots & \vdots & \vdots \\ \hline \alpha^r_{m1} & \dots & \alpha^r_{mt} & \dots & \alpha^r_{mn} \\ \alpha^g_{m1} & \dots & \alpha^g_{mt} & \dots & \alpha^g_{mn} \end{array}\right] \left(, \; BWO^r_{iter} = \left[\begin{array}{c} \alpha^r_{r1} \\ \alpha^g_{r1} \\ \hline \vdots \\ \hline \alpha^r_{rs} \\ \alpha^g_{rs} \\ \hline \vdots \\ \hline \alpha^r_{rn} \\ \alpha^g_{rn} \end{array}\right] \right) \tag{5}$$

From matrices M^c_{iter} and M^r_{iter}, finding any terminal edges that exist in the current iteration (which corresponds to lines 5 and 6 in Algorithm 1) will entail three logical operations performed in sequence: (i) Bit-Wise-Or (BWO) in Equations 6 and 7, (ii) eXclusive-OR (XOR) in Equations 8 and 9, and (iii) OR operation in Equations 10. Two parallel BWO operations are derived as shown in Equations 6 and 7.

$$BWO^{c}_{iter} = \begin{bmatrix} (\alpha^r_{c1}, \alpha^g_{c1}) & (\alpha^r_{c2}, \alpha^g_{c2}) & \cdots & (\alpha^r_{ct}, \alpha^g_{ct}) & \cdots & (\alpha^r_{cn}, \alpha^g_{cn}) \end{bmatrix} \tag{6}$$

where $\forall t$, $\alpha^r_{ct} = \bigvee_{s=1}^{m} \alpha^r_{st}$ and $\alpha^g_{ct} = \bigvee_{s=1}^{m} \alpha^g_{st}$, and notation $\bigvee$ means Bit-Wise-Or of elements.

$$BWO^{r}_{iter} = \begin{bmatrix} (\alpha^r_{r1}, \alpha^g_{r1}) & (\alpha^r_{r2}, \alpha^g_{r2}) & \cdots & (\alpha^r_{rs}, \alpha^g_{rs}) \cdots & (\alpha^r_{rn}, \alpha^g_{rn}) \end{bmatrix}^T \tag{7}$$

where $\forall s$, $\alpha^r_{rs} = \bigvee_{t=1}^{n} \alpha^r_{st}$ and $\alpha^g_{rs} = \bigvee_{t=1}^{n} \alpha^g_{st}$.

The XOR operations are as follows.

$$XOR^{c}_{iter} = \begin{bmatrix} \tau_{c1} & \tau_{c2} & \cdots & \tau_{ct} & \cdots & \tau_{cn} \end{bmatrix} \tag{8}$$

where $\forall t$, $\tau_{ct} = \alpha^r_{ct} \oplus \alpha^g_{ct}$ and $\oplus$ denotes eXclusive-OR.

$$XOR^{r}_{iter} = \begin{bmatrix} \tau_{r1} & \tau_{r2} & \cdots & \tau_{rs} & \cdots & \tau_{rm} \end{bmatrix}^T \tag{9}$$

where $\forall s$, $\tau_{rs} = \alpha^r_{rs} \oplus \alpha^g_{rs}$. Equation 8 shows the existence of terminal nodes in any column, and Equation 9, in any row.

Among the sequence of three logical operations, Equation 10 shows the OR operation that produces the termination condition (i.e., the further reducibility of matrix M_{iter}, which corresponds to line 7 in Algorithm 1) at each iteration. If T_{iter} is one, i.e., logically true, then more terminal edges exist; thus, further iterations must continue. However, if the current matrix is irreducible (i.e., it has no terminal edges), T_{iter} will become '0'; thus, further iterations would accomplish nothing.

$$T_{iter} = \tau_C \vee \tau_R \tag{10}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{ct}$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{rs}$. The next iteration M_{iter+1} is derived from Equations 8 and 9 according to the following criterion (which corresponds to lines 8 and 9 of Algorithm 1).

$$(\alpha^r_{st}, \alpha^g_{st})_{k+1} = \begin{cases} (\alpha^r_{st}, \alpha^g_{st})_k, & \text{if } \tau_{ct} = 0 \text{ and } \tau_{rs} = 0 \\ (0,0), & \text{if } \tau_{ct} = 1 \text{ or } \tau_{rs} = 1 \end{cases} \tag{11}$$

where k refers to k^{th} iteration, and $k+1$ refers to $(k+1)^{\text{th}}$ iteration. That is, the next iteration will begin with a new matrix M_{iter+1} calculated by Equation 11 from M_{iter}.

Before finishing PDDA, however, one more important process remains: deadlock detection, which requires two more parallel logic operations shown in Equations 12 and 13. Equation 12 represents the existence of connect nodes involved in a cycle in any column, and Equation 13, in any row.

$$AND^{c}_{iter} = \begin{bmatrix} \phi_{c1} & \phi_{c2} & \cdots & \phi_{ct} & \cdots & \phi_{cn} \end{bmatrix} \tag{12}$$

where $\forall t$, $\phi_{ct} = \alpha^{r}_{ct} \wedge \alpha^{g}_{ct}$ and $\wedge$ denotes bit-wise-and.

$$AND^{r}_{iter} = \begin{bmatrix} \phi_{r1} & \phi_{r2} & \cdots & \phi_{rs} & \cdots & \phi_{rm} \end{bmatrix}^{T} \tag{13}$$

where $\forall s$, $\phi_{rs} = \alpha^{r}_{rs} \wedge \alpha^{g}_{rs}$ and $\wedge$ denotes bit-wise-and.

Finally, Equation 14 produces the result of deadlock detection, which corresponds to lines 8-12 of Algorithm 2.

$$D_{iter} = \phi_c \vee \phi_r, \qquad \text{when } T_{iter} = 0 \tag{14}$$

where $\phi_c = \bigvee_{t=1}^{n} \phi_{ct}$ and $\phi_r = \bigvee_{s=1}^{m} \phi_{rs}$.

We define two more equations that are used to describe the DDU architecture in Section 7.3. From Equations 8 and 12, we define a column weight vector as follows:

$$W^{c}_{iter} = \begin{bmatrix} w_{c1} & w_{c2} & \cdots & w_{ct} & \cdots & w_{cn} \end{bmatrix} \tag{15}$$

where n is the number of processes, and $\forall t$, w_{ct} is a pair (τ_{ct}, ϕ_{ct}) representing whether the corresponding process node is a terminal node, a connect node, or neither.

From Equations 9 and 13, we define a row weight vector as follows:

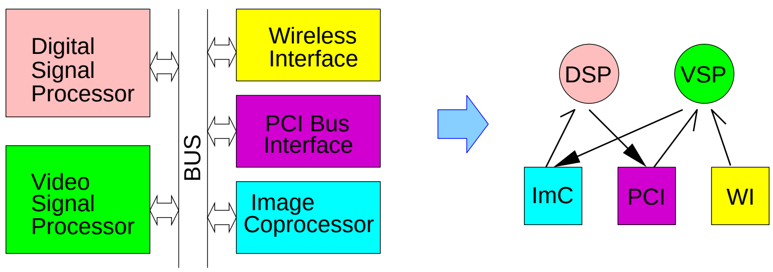
$$W^{r}_{iter} = \begin{bmatrix} w_{r1} & w_{r2} & \cdots & w_{rs} & \cdots & w_{rm} \end{bmatrix}^{T} \tag{16}$$

where m is the number of resources, and $\forall s$, w_{rs} is a pair (τ_{rs}, ϕ_{rs}) representing whether the corresponding resource node is a terminal node, a connect node, or neither.

7.2. DDU Operation Examples

We now illustrate a series of logical operations of the DDU with two simple examples. In one example, the current state of a system consisting of two processes and three resources has a deadlock. Whereas, in the other example, the current state of the same system does not have a deadlock.

Example 13. Two Processes and Three Resources with a Cycle



(a) an SoC (b) a RAG

FIGURE 14. SoC example

An SoC shown in Figure 14 has two processes, p_1 running on DSP and p_2 running on VSP, and three resources, ImC, PCI and WI as q_1, q_2 and q_3, respectively. The matrix representation of this example is shown in Figure 15.

P \ Q	p_1(DSP)	p_2(VSP)
q_1(ImC)	g	r
q_2(PCI)	r	g
q_3(WI)	0	g

FIGURE 15. SoC example with two processors and three resources with a cycle.

We now show how the parallel operation of the DDU works cycle by cycle. In the beginning, given γ_{ij} shown in Figure 14 (b), lines 2-6 of Algorithm 2 construct an initial matrix M_{ij} shown in Equation 17. Each element in M_{ij} is referred to α_{st}, where $1 \leq s \leq m$ and $1 \leq t \leq n$. Here, each α_{st} can have a value either g, r or 0.

$$M_{ij} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} g & r \\ r & g \\ 0 & g \end{bmatrix} = M_{iter1} \text{ (at the first iteration)} \tag{17}$$

Then, line 7 of Algorithm 2 calls Algorithm 1. However, instead of the call, M_{ij} just becomes M_{iter1} by a detection start signal. Since M_{iter1} needs to be processed in a two-dimensional operation, we represent M_{iter1} as two-bit binary encoding column and row representations, M^c_{iter1} and M^r_{iter1}, respectively, which are shown in Equation 18. Again, note that these two matrices are the same as M_{iter1} and they are only shown for the purpose of the understanding of the parallel two-dimensional operation inside the DDU.

$$M^c_{iter1} = \begin{bmatrix} 01 & 10 \\ 10 & 01 \\ 00 & 01 \end{bmatrix}, \; M^r_{iter1} = \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 0 & 0 \\ 0 & 1 \end{array}\right] \tag{18}$$

Now a Bit-Wise-Or ($BWO, \vee$) operation is applied to each column of M^c_{iter1} and makes the BWO^c_{iter1} matrix, shown in Equation 19. Similarly, another BWO operation is applied to each row of M^r_{iter1} and makes the BWO^r_{iter1} matrix, shown in Equation 19. These BWO operations are processed in parallel and the results are fed to the next operation to reveal both terminal rows and terminal columns.

$$BWO^c_{iter1} = \begin{bmatrix} 11 & 11 \end{bmatrix},\ BWO^r_{iter1} = \left[\begin{array}{c} 1 \\ 1 \\ \hline 1 \\ 1 \\ \hline 0 \\ 1 \end{array}\right] \tag{19}$$

Next, an eXclusive-OR (XOR, $\oplus$) operation is applied to both bits of each entry of BWO^c_{iter1} and makes the XOR^c_{iter1} matrix, shown in Equation 20. For instance, if $BWO^c = \begin{bmatrix} 11 & 01 \end{bmatrix}$, then XOR^c will be $\begin{bmatrix} 1 \oplus 1 & 0 \oplus 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix}$. A similar XOR operation is also applied to both bits of each entry of BWO^r_{iter1} and produces the XOR^r_{iter1} matrix, shown in Equation 20.

$$XOR^c_{iter1} = \begin{bmatrix} \tau_{c1} & \tau_{c2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix},\ XOR^r_{iter1} = \begin{bmatrix} \tau_{r1} & \tau_{r2} & \tau_{r3} \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \tag{20}$$

An element '1' in XOR^c_{iter1} represents that the corresponding column (recall a column corresponds to a process) in M_{iter1} is a terminal process node; thus, the column is reducible. On the other hand, an element '1' in XOR^r_{iter1} indicates that the corresponding row (recall a row corresponds to a resource) in M_{iter1} is a terminal resource node; hence, the row is reducible. Since the element of the third row in XOR^r_{iter1} is '1,' which signifies the third row is a terminal row in matrix M_{iter1}, the third row can be eliminated from further consideration. That is, more iterations need to continue, which is shown in Equation 21.

$$T_{iter1} \quad = \tau_C \vee \tau_R \quad = 1 \tag{21}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{ct} = 0$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{rs} = 1$. Since T_{iter1} in Equation 21 results in '1,' there exist terminal edges at this iteration; thus, further iteration(s) are necessary. Before continuing the next iteration, we calculate connect nodes, AND^c_{iter1} and AND^r_{iter1}, shown in Equation 22.

$$AND^c_{iter1} = \begin{bmatrix} \phi_{c1} & \phi_{c2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix},\ AND^r_{iter1} = \begin{bmatrix} \phi_{r1} & \phi_{r2} & \phi_{r3} \end{bmatrix}^T = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T \tag{22}$$

Weight vectors are W^c_{iter1} and W^r_{iter1}, shown in Equation 23.

(23)
$$W^c_{iter1} = \begin{bmatrix} w_{c1} & w_{c2} \end{bmatrix} = \begin{bmatrix} (0,1) & (0,1) \end{bmatrix},\ W^r_{iter1} = \begin{bmatrix} w_{r1} & w_{r2} & w_{r3} \end{bmatrix}^T = \begin{bmatrix} (0,1) & (0,1) & (1,0) \end{bmatrix}^T$$

where (0,1) signifies a connect node and (1,0) signifies a terminal node. Thus, row 3 is a terminal node. After terminal edges revealed in iteration one are eliminated by Equation 11, the next iteration begins with new M_{iter2} shown in Equation 24.

$$M_{iter2} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} g & r \\ r & g \\ 0 & 0 \end{bmatrix} \text{ (at the second iteration)} \tag{24}$$

M^c_{iter2} and M^r_{iter2} at iteration 2 are shown in Equation 25.

$$M^c_{iter2} = \begin{bmatrix} 01 & 10 \\ 10 & 01 \\ 00 & 00 \end{bmatrix},\ M^r_{iter2} = \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 0 & 0 \\ 0 & 0 \end{array}\right] \tag{25}$$

Then, BWO^c_{iter2} and BWO^r_{iter2} are shown in Equation 26.

$$BWO^c_{iter2} = \left[\begin{array}{cc} 11 & 11 \end{array}\right],\ BWO^r_{iter2} = \left[\begin{array}{c} 1 \\ 1 \\ \hline 1 \\ 1 \\ \hline 0 \\ 0 \end{array}\right] \tag{26}$$

Next, XOR^c_{iter2} and XOR^r_{iter2} are shown in Equation 27, which has no terminal nodes.

$$XOR^c_{iter2} = \left[\begin{array}{cc} 0 & 0 \end{array}\right],\ XOR^r_{iter2} = \left[\begin{array}{c} 0 \\ \hline 0 \\ \hline 0 \end{array}\right] \tag{27}$$

As a result, T_{iter2} becomes '0' as shown in Equation 28, which means M_{iter2} is irreducible.

$$T_{iter2} \quad = \tau_C \vee \tau_R \quad = 0 \tag{28}$$

since $\tau_C = \bigvee_{t=1}^{n} \tau_{ct} = 0$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{rs} = 0$. Therefore, this iteration is the last. At this moment, we need to find D_{iter2}, the deadlock decision result. To do this, we first calculate AND^c_{iter2} and AND^r_{iter2}, as shown in Equation 29, which represent the existence of connect nodes in columns and in rows, respectively.

$$AND^c_{iter2} = \left[\begin{array}{cc} 1 & 1 \end{array}\right] \text{ and } AND^r_{iter2} = \left[\begin{array}{ccc} 1 & 1 & 0 \end{array}\right]^T \tag{29}$$

Weight vectors at this iteration are W^c_{iter2} and W^r_{iter2}, shown in Equation 30.

$$W^c_{iter2} = \left[\begin{array}{cc} (0,1) & (0,1) \end{array}\right],\ W^r_{iter2} = \left[\begin{array}{ccc} (0,1) & (0,1) & (0,0) \end{array}\right]^T \tag{30}$$

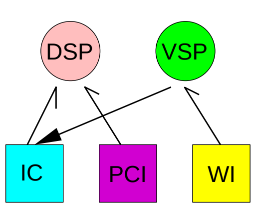
where (0,1) signifies a connect node and (0,0) in W^r_{iter2} signifies no edges in the third row. Next, the decision of a deadlock is made by Equation 31.

$$D_{iter2} \quad = \phi_c \vee \phi_r = 1, \qquad \text{when } T_{iter2} = 0 \tag{31}$$

where $\phi_c = \bigvee_{t=1}^{n} \phi_{ct} = 1$ and $\phi_r = \bigvee_{s=1}^{m} \phi_{rs} = 1$. Since (i) $T_{iter2} = 0$, which signifies that this new $M_{i,j+1}$ is not reducible any more, and (ii) $D_{iter2} = 1$, which signifies that edges still exist, we finally find that γ_{ij} has a deadlock. □

Example 14. Two Processes and Three Resources without a Cycle

Consider the same system, shown in Example 13. However, the system currently has a different set of request and grant edges, as shown in Figure 16.



(a) a RAG

P \ Q	p_1(DSP)	p_2(VSP)
q_1(ImC)	g	r
q_2(PCI)	g	0
q_3(WI)	0	g

(b) M_{ij}

FIGURE 16. An SoC example with two processors and three resources without a cycle.

We now show how the parallel operation of the DDU works cycle by cycle in this case, which has no cycles. First, given γ_{ij} shown in Figure 16 (a), initial matrix M_{ij} shown below is constructed by lines 2-6 of Algorithm 2.

$$M_{ij} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} g & r \\ g & 0 \\ 0 & g \end{bmatrix} = M_{iter1} \text{ (at the first iteration)} \tag{32}$$

Second, two corresponding binary coded matrices M^c_{iter1} and M^r_{iter1}, shown in Equation 33, are constructed according to the binary encoding scheme, explained in Section 7.1.

$$M^c_{iter1} = \begin{bmatrix} 01 & 10 \\ 01 & 00 \\ 00 & 01 \end{bmatrix}, \; M^r_{iter1} = \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ \hline 0 & 0 \\ 1 & 0 \\ \hline 0 & 0 \\ 0 & 1 \end{array}\right] \tag{33}$$

Third, a Bit-Wise-Or operation is applied to each column of M^c_{iter1}, resulting in the BWO^c_{iter1} matrix shown in Equation 34. At the same time, a similar Bit-Wise-Or operation is applied to each row of M^r_{iter1}, resulting in the BWO^r_{iter1} matrix.

$$BWO^c_{iter1} = \begin{bmatrix} 01 & 11 \end{bmatrix}, \; BWO^r_{iter1} = \left[\begin{array}{c} 1 \\ 1 \\ \hline 0 \\ 1 \\ \hline 0 \\ 1 \end{array}\right] \tag{34}$$

Fourth, an eXclusive-OR($\oplus$) operation is applied to the two bits of each entry of BWO^c_{iter1}, resulting in the XOR^c_{iter1} matrix, shown in Equation 35. Likewise, another eXclusive-OR operation is also applied to the two bits of each entry of BWO^r_{iter1}, resulting in the XOR^r_{iter1} matrix.

$$XOR^c_{iter1} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; XOR^r_{iter1} = \left[\begin{array}{c} 0 \\ \hline 1 \\ \hline 1 \end{array}\right] \tag{35}$$

Now, the termination condition is calculated by Equation 36.

$$T_{iter1} \quad = \tau_C \vee \tau_R \quad = 1 \tag{36}$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{ct} = 1$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{rs} = 1$. Since T_{iter1} in Equation 36 results in '1,' there exists at least one more terminal edge at this iteration; thus, at least one more iteration is necessary.

In Equation 35, since the element of the first column in XOR^c_{iter1} is '1,' which means the first column is a terminal column in matrix M_{iter1}, the first column can be eliminated from further consideration. Also, since the elements of second and third rows in XOR^r_{iter1} are '1,' which means the second and third rows are terminal rows in matrix M_{iter1}, the second and third rows can be eliminated from further consideration. After the terminal edges revealed in iteration one are eliminated using Equation 11, the next iteration begins with new M_{iter2}, shown in Equation 37.

$$M_{iter2} = \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \\ \alpha_{31} & \alpha_{32} \end{bmatrix} = \begin{bmatrix} 0 & r \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 00 & 10 \\ 00 & 00 \\ 00 & 00 \end{bmatrix} \tag{37}$$

BWO^c_{iter2}, XOR^c_{iter2}, BWO^r_{iter2} and XOR^r_{iter2} are shown in Equations 38 and 39, respectively.

$$(38) \qquad BWO^{c}_{iter2} = \left[\begin{array}{cc} 00 & 10 \end{array} \right], \; BWO^{r}_{iter2} = \left[\begin{array}{c} 1 \\ 0 \\ \hline 0 \\ 0 \\ \hline 0 \\ 0 \end{array} \right]$$

$$(39) \qquad XOR^{c}_{iter2} = \left[\begin{array}{cc} 0 & 1 \end{array} \right], \; XOR^{r}_{iter2} = \left[\begin{array}{c} 1 \\ \hline 0 \\ \hline 0 \end{array} \right]$$

The termination condition is shown in Equation 40.

$$(40) \qquad T_{iter2} \quad = \tau_C \vee \tau_R \quad = 1$$

where $\tau_C = \bigvee_{t=1}^{n} \tau_{ct} = 1$ and $\tau_R = \bigvee_{s=1}^{m} \tau_{rs} = 1$. After the terminal edge revealed at the second iteration is eliminated by Equation 11, the next iteration begins with new M_{iter3}, shown in Equation 41.

$$(41) \qquad M_{iter3} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Since this M_{iter3} has no edges, all the remaining results of the deadlock equations will turn out to be '0'; thus, the DDU identifies that the system state γ_{ij} does not have a deadlock. $\square$

7.3. DDU Architecture

This subsection describes the architecture of the DDU. The architecture of the DDU consists of three parts as shown in Figure 17: matrix cells, weight cells and a decide cell. Part 1 is the system state matrix M_{ij} consisting of an array of matrix cells α_{st} that represent $(\alpha^r_{st}, \alpha^g_{st})$. Part 2 consists of two weight vectors: (i) one column weight vector W^c (shown in Equation 15) below the matrix cells and (ii) one row weight vector W^r (shown in Equation 16) on the right side of matrix cells. Each element w_{ct} in W^c is called a column weight cell, and each element w_{rs} in W^r is called a row weight cell. Part 3 is one decide cell D_{iter} at the bottom right corner of the DDU. All cells are interconnected via buses.

Figure 17 shows the architecture of the DDU for three processes and three resources. This DDU has nine matrix cells (3×3) for each edge element of M_{ij}, six weight cells (three for column processing and three for row processing), and one decide cell for making the decision of a deadlock.

The matrix cell shown in Figure 18, which corresponds to each α_{st}, has one 2-bit register as a core with two input and four output signals. One pair of outputs (C^r_{st}, C^g_{st}) goes to a column weight cell w_{ct} and is used to determine a terminal or connect process node. The other pair of outputs (R^r_{st}, R^g_{st}) goes to a row weight cell w_{rs} and is used to determine a terminal or connect resource node. Note that C^r_{st} and R^r_{st} are the same, and C^g_{st} and R^g_{st} are the same. Two inputs

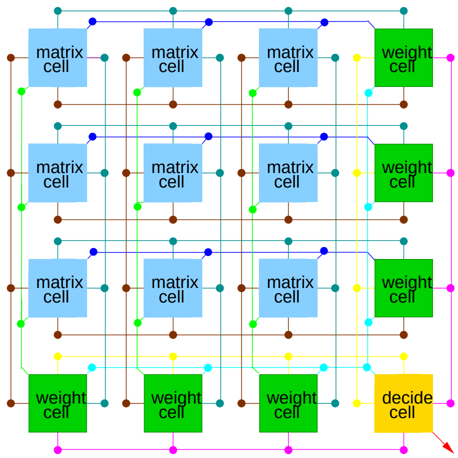


FIGURE 17. DDU architecture

(τ_{rs} and τ_{ct}) from weight cells are used to suppress outputs to emulate the terminal reduction when this cell belongs to a terminal node (i.e., when this matrix cell corresponds to a terminal edge, this cell must be excluded from further iteration).

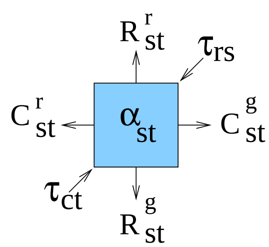


FIGURE 18. Matrix Cell α_{st} in a matrix array M_{ij}

Figure 19 depicts the logic diagram of the matrix cell for row one and column one of Example 13. All outputs are controlled by the OR of τ_{c1} and τ_{r1}, which indicate that the corresponding column or row is a terminal node.

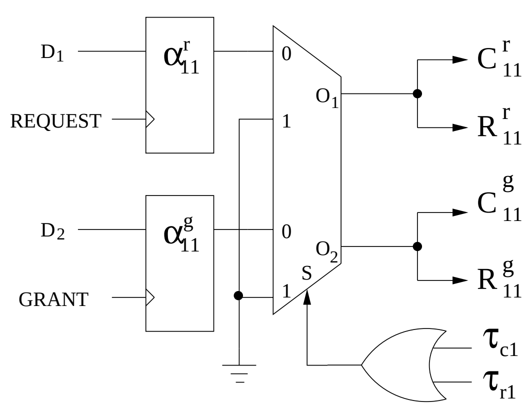
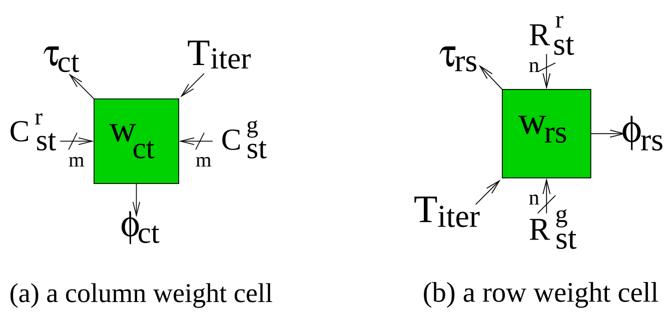


FIGURE 19. The logic diagram of a matrix cell

The weight cell shown in Figure 20 is a mathematical calculation unit that performs a series of logic operations with outputs of all matrix cells in a row or in a column simultaneously to reveal a terminal node as described in Section 7.1. All column weight cells calculate BWO^c_{iter} in Equation 6, XOR^c_{iter} in Equation 8, and AND^c_{iter} in Equation 12 to reveal all terminal process nodes. All row weight cells, in contrast, calculate BWO^r_{iter} in Equation 7, XOR^r_{iter} in Equation 9, and AND^r_{iter} in Equation 13 to reveal all terminal resource nodes. The weight cell has a flip-flop (FF) that stores the result of each iteration. The m pairs of inputs (C^r_{st}, C^g_{st}) of w_{ct}, shown in Figure 20 (a), all come from matrix cells in column t, and the n pairs of inputs (R^r_{st}, R^g_{st}) of w_{rs}, shown in Figure 20 (b), all come from matrix cells in row s, where m and n are the numbers of resources and processes, respectively. A pair of outputs (τ_{ct}, ϕ_{ct}) of w_{ct} is an element w_{ct} of W^c, shown in Equation 15. Also, a pair of outputs (τ_{rs}, ϕ_{rs}) of w_{rs} is an element w_{rs} of W^r, shown in Equation 16.



(a) a column weight cell w_{ct} in W^c (b) a row weight cell w_{rs} in W^r

FIGURE 20. Weight Cells in W^c and W^r described in Equations 15 and 16

Figure 21 depicts the logic diagram of the weight cell for the first column of Example 13. The weight cell for the second column of Example 13 would be exactly the same logic but with inputs C^r_{12}, C^r_{22} and C^r_{32} to the top OR gate, inputs C^g_{12}, C^g_{22} and C^g_{32} to the bottom OR gate, and a pair of outputs (τ_{c2}, ϕ_{c2}).

The decide cell shown in Figure 22 is a decision unit that gathers outputs from all column and row weight cells and generates two signals: T_{iter} (shown in Equations 10), which tells whether M_{iter} is reducible further or not, and D_{iter} (shown in Equations 14), which tells whether the system state has a deadlock or not when $T_{iter} = 0$.

Figure 23 depicts the logic diagram of the decide cell for Example 13. The three inputs ϕ_{r1}, ϕ_{r2} and ϕ_{r3} come from three row weight cells, and two inputs ϕ_{c1} and ϕ_{c2} come from two column weight cells. Similarly, the three inputs τ_{r1}, τ_{r2} and τ_{r3} come from three row weight cells, and two

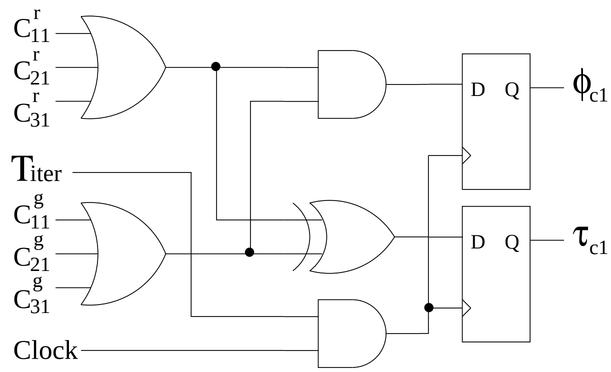


FIGURE 21. The logic diagram of a weight cell

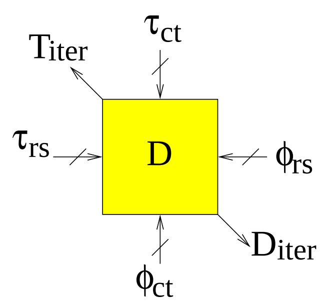


FIGURE 22. Decide Cell D

inputs τ_{c1} and τ_{c2} come from two column weight cells. One output D_{iter}, which tells the existence of connect nodes, is the result of deadlock detection, and the other output T_{iter} tells the existence of terminal edges.

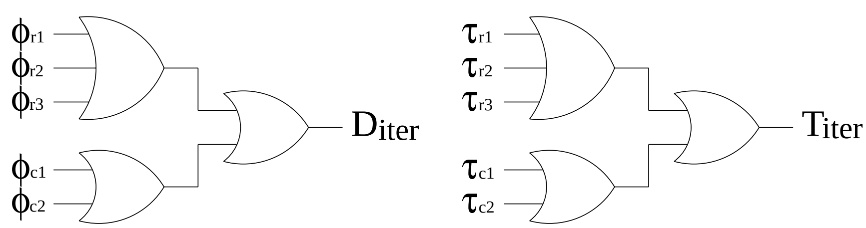


FIGURE 23. The logic diagram of a decide cell

7.4. Synthesis Result of the Deadlock Detection Unit

We used the Synopsys Design Compiler (DC) to synthesize the DDU with a $0.3\mu m$ standard cell library from AMIS [10]. Table 1 shows the synthesis results of five types of DDUs customized according to the number of processes and resources in an SoC. The fourth column, denoted "logic delay per iteration," represents a logic delay per each iteration for the corresponding DDU. The fifth column, denoted "worst case # iterations," represents the number of worst case number of iterations for the corresponding DDU. The sixth column, denoted "worst case delay," results from "logic delay per one iteration" multiplied by "worst case # iterations." Table 1 reveals the following. (i) The worst case number of iterations increases linearly with the smaller number of the number of processes and the number of resources. (ii) The logic delay per iteration increases proportional to the larger number of the number of processes and the number of resources. (iii) The other numbers in Table 1 increase almost quadratically proportional to the total number of processes plus resources.

# processes × # resources	lines of Verilog	area in terms of two-input NAND gates	logic delay per iteration (ns)	worst case # iterations	the worst case delay (ns)
2×3	49	186	0.91	2	1.82
5×5	73	364	2.21	6	13.26
7×7	102	455	2.51	10	25.1
10×10	162	622	3.66	16	58.56
50×50	2682	14142	4.12	96	395.52

Table 1. Synthesis results of DDU's

8. Experimental Setup and Results

8.1. Experimental Setup

In the experiments, we implemented in Verilog HDL the SoC architecture shown in Figure 24 (except PE cores, which are typically provided by simulation tool vendors such as processor support packages from Seamless CVE [13]). The SoC has four Motorola MPC755s as processing elements (PEs). The MPC755 has two separate instruction and data L1 caches each of size 32KB. The SoC has also four resources: a video interface (VI) device, a DSP, an MPEG processor and a wireless interface (WI) device. These four resources have timers, interrupt generators and input/output ports that are necessary to support our simulations. In addition, the SoC has a DDU for five processes and five resources, an arbiter and 16MB of shared memory. The master clock rate of the bus system is 10ns. However, MPC755 runs at the 400MHz clock-rate internally [16].

We assumed the following in our experiments. (i) The SoC is capable of capturing still and motion pictures, processing MPEG encoding and decoding, and performing image signal processing. (ii) The device can also support data streaming applications using a standard wireless LAN card. (iii) Such functionalities are implemented in hardware in this SoC; thus, each PE will request the services of the hardware units.

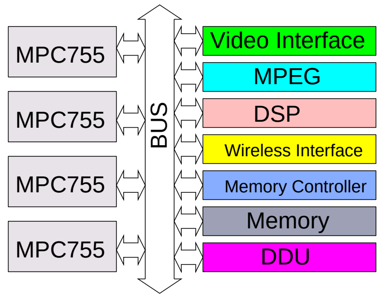


FIGURE 24. Experimental SoC architecture

To benefit from a real-time operating system (RTOS), we used Atalanta RTOS version 0.3 [17], a shared-memory multiprocessor RTOS developed at the Georgia Institute of Technology. The RTOS code resides in the shared memory, and all PEs execute the same RTOS code and share kernel structures and the states of all processes and resources.

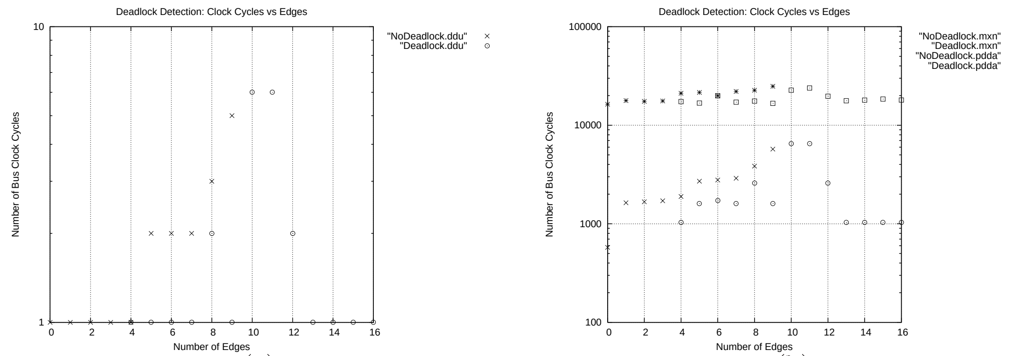
On top of the SoC with the Atalanta RTOS, we completed two experiments with three deadlock detection implementations: (i) the DDU, (ii) a software implementation of PDDA and (iii) a software implementation of an $O(m \times n)$ deadlock detection algorithm.[4] One was a performance comparison among the three implementations while the other experiment was a comparison of the execution time of an application using DDU hardware (i) versus using the faster software deadlock detection algorithm (ii). We accomplished the two experiments through cycle-accurate simulations. The simulations were carried out using Mentor Graphics' Seamless Co-Verification Environment (CVE) [13], aided by Synopsys' VCS$^{\text{TM}}$ [14] for Verilog HDL simulation and XRAY$^{\text{TM}}$ [15] for software debugging.

8.2. **Execution Time Comparison of PDDA**

In the first experiment, with the SoC described in Section 8.1, we simulated a large number of deadlock detection cases with a various numbers of request and grant edges (up to sixteen) to measure the execution time difference among the three implementations. The experimental results are shown in Figure 25.

All simulations were executed in a system having five resources and five processes. In the legend of Figure 25, "NoDeadlock.ddu," "NoDeadlock.pdda" and "NoDeadlock.mxn" represent non-deadlocked scenarios checked by the DDU, PDDA in software and an $O(m \times n)$ deadlock detection algorithm in software, respectively. By contrast, "Deadlock.ddu," "Deadlock.pdda" and "Deadlock.mxn" represent deadlocked scenarios checked by the three implementations, respectively. Figure 25 demonstrates, in all cases, (i) three orders of magnitude difference in the number

[4]Algorithm 4.4 in [7]



(a) (b)

FIGURE 25. The execution-time performance of deadlock detection using PowerPC's

of cycles between the DDU and the other software implementations and (ii) about one order of magnitude difference in average between PDDA and the $O(m \times n)$ software algorithm. The reason is that while PDDA in software does bit-wise operations, the $O(m \times n)$ software algorithm executes a lot of instructions to travel nodes, search linked lists and update data structures. Note that this comparison is only a part of a bigger picture, which we will show in the next experiment.

8.3. Execution Time Comparison of an Application

In the second experiment, we wanted to identify the difference in an application executing using the DDU versus PDDA in software (note that our software version of PDDA executes much faster than the $O(m \times n)$ deadlock detection algorithm[5] in software as shown in Figure 25). We devised an application example inspired by the Jini lookup service system [12], in which client applications can request services through intermediate layers (i.e., lookup, discovery and admission). Since the SoC, introduced in Section 8.1, has multiple processes and multiple resources, and Assumptions 2-4 in Section 5 can also easily be satisfied during the normal execution of the application, a deadlock is possible in such a system. Thus, this is an example of a practical application that can benefit from the DDU. In this experiment, we invoked one process on each PE and prioritized all processes, p_1 being the highest and p_4 being the lowest.

We show a sequence of requests and grants that finally leads to a deadlock as shown in Table 2 and Figure 26. Process p_1, running on PE1, requests both the MPEG and the VI at time t_1, which are then granted to p_1. After that, p_1 starts receiving a video stream through the VI and does MPEG processing. At time t_2, process p_3, running on PE3, needs and requests the MPEG and the WI to simultaneously convert a frame to an image and send the image through the WI. However, only the WI is granted to p_3 since the MPEG is unavailable. At time t_3 shown in Table 2

[5]Algorithm 4.4 in [7]

and Figure 26, p_2 running on PE2 also requests the MPEG and WI hardware units, which are not available for p_2. When the MPEG is released by p_1 at time t_4, the MPEG is granted to p_2 since p_2 has a higher priority than p_3. This last grant will lead to a deadlock in the SoC.

Time	Events No.	Events
t_0	e_0	The application starts.
t_1	e_1	p_1 requests MPEG and VI; MPEG and VI are granted to p_1 immediately.
t_2	e_2	p_3 requests MPEG and WI; WI is granted to p_3 immediately.
t_3	e_3	p_2 requests MPEG and WI. Both p_2 and p_3 wait MPEG.
t_4	e_4	MPEG is released by p_1.
t_5	e_5	MPEG is granted to p_2. since p_2 has a higher priority than p_3.

TABLE 2. A sequence of requests and grants that leads to deadlock

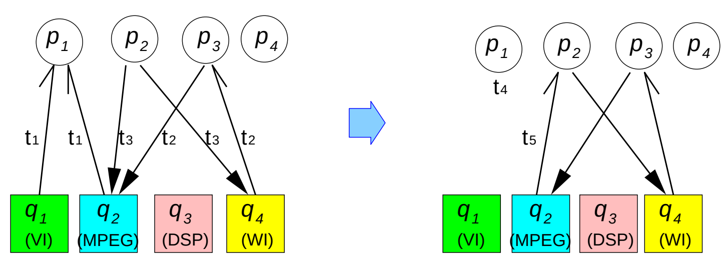


FIGURE 26. Events RAG

With the above scenario, we measured both the deadlock detection time Δ and the application execution time from the application start (t_0) until the detection of a deadlock in two cases: using (i) the DDU and (ii) PDDA in software. Note that the RTOS initialization time was excluded (i.e., the RTOS is assumed to be fully operational at time t_0). Table 3 shows that (i) in average the DDU achieved a 1408X speedup over the software implementation of PDDA and that (ii) the DDU gave a 46% speed-up of application execution time over PDDA in software.

Method of Deadlock Detection	Algorithm Execution Time (Δ) in Cycles	Application Execution Time in Cycles
DDU	1.3 (average)	27714
PDDA in software	1830 (average)	40523

* Deadlock checks were executed 10 times during the entire run-time of the application

TABLE 3. Deadlock detection time and application execution time

9. **Conclusions**

This paper describes a novel parallel deadlock detection algorithm called PDDA and illustrates a hardware implementation of PDDA called Deadlock Detection Unit (DDU). We provide detailed operation examples and synthesis results. Moreover, this paper proves the correctness of PDDA and the run-time complexity of the DDU, which is $2 \times min(m,n) - 3 = O(min(m,n))$.

We experimented two sets of performance comparisons for the evaluation of this research. Both experiments were carried out through cycle-accurate simulations of a System-on-a-Chip (SoC) that consists of four Motorola MPC755s (each of which has 32KB separate instruction and data caches), a DDU, 16MB of shared memory and four types of resources.

The first experiment demonstrated that the DDU reduces the execution time of deadlock detection by 99.9% *(i.e., 1000X)* or more as compared to two software implementations of deadlock detection algorithms. The second experiment showed that the DDU provide an application that exploits the DDU with a 46% speed-up in the execution time from the application start until the detection of a deadlock over the case in which the same application uses a software implementation of PDDA.

The two experimental results substantiate the following advantages of this research. The DDU can provide developers with (i) a system that can check for deadlock more frequently (so far, deadlock detection in a real-time system has been out of consideration due to the long execution time of software deadlock detection that inevitably entails performance degradation), (ii) a system that has more design flexibility due to the time gained from using the DDU instead of software deadlock detection and (iii) a system that has a deadlock detection capability with a nearly zero performance penalty which is significant since all deadlock prevention methods proposed in previous work have much higher performance penalties [7].

Acknowledgments

This research has been funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge the help of Kyeong Keol Ryu, who proofread multiple versions of this manuscript. We also would like to acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, National Semiconductor, Sun Microsystems and Synopsys.

References

[1] The International Technology Roadmap for Semiconductors 2002, http://public.itrs.net/.

[2] E. Coffman, M. Elphick and A. Shoshani, "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, June 1971.

[3] A. Shoshani and E. Coffman Jr., "Detection, Prevention and Recovery From Deadlocks in Multiprocess, multiple resource systems," *4th Annual Princeton Conf. on Information Sciences and System*, March 1970.

[4] R. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing surveys*, vol. 4, no. 3, pp.179-196, September 1972.

[5] T. Leibfried Jr., "A Deadlock Detection and Recovery Algorithm Using the Formalism of a Directed Graph Matrix," *Operation Systems Review*, vol. 23, no. 2, pp. 45-55, April 1989.

[6] J. Kim and K. Koh, "An O(1) Time Deadlock Detection Scheme in Single Unit and Single Request Multiprocess System," *IEEE TENCON '91*, vol 2, pp. 219-223, August 1991.

[7] M. Maekawa, A. Oldhoeft and R. Oldehoeft, *Chapter 4 of Operating Systems - Advanced Concepts*, Benjamin/Cummings Publishing Company, Menlo Park, CA, 1987.

[8] O. Coudert, J. Madre and H. Fraisse, "A New Viewpoint on Two-Level Logic Minimization," *Proceedings of 30th Design Automation Conference*, pp. 625–630, June 1993.

[9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994.

[10] AMI Semiconductor, http://www.amis.com.

[11] P. Shiu, Y. Tan and V. Mooney, "A Novel Parallel Deadlock Detection Algorithm and Architecture," *9th International Workshop on Hardware/Software Co-Design (CODES'01)*, pp. 30-36, April 2001.

[12] S. Morgan, "Jini to the rescue," *IEEE Spectrum*, April 2000.

[13] Mentor Graphics, Hardware/Software Co-Verification: Seamless, http://www.mentor.com/seamless/.

[14] Synopsys VCS™ Verilog Simulator, http://www.synopsys.com/products/simulation/simulation.html.

[15] Mentor Graphics, XRAY® Debugger, http://www.mentor.com/xray/.

[16] Motorola, http://e-www.motorola.com/.

[17] D. Sun, D. Blough and V. Mooney, "Atalanta: A new multiprocessor RTOS kernel for System-on-a-Chip Applications," Tech. Rep. GIT-CC-02-19, http://www.cc.gatech.edu/pubs.html, 2002.

[18] R. Holt. *On Deadlock in Computer Systems*, Ph.D. Dissertation, Cornell University, Ithaca, NY, January 1971.

[19] E. Brauoe, "An Algorithm for The Detection of System Deadlocks," IBM Technical Report: TROO-791, IBM Data Systems Division, Poughkeepsie, NY, 1961.

[20] E. Dijkstra, "Cooperating Sequential Processes," Tech. Rep. Technological University, Eindhoven, The Netherlands, September 1965.

[21] N. Kaveh and W. Emmerich, "Deadlock Detection in Distribution Object Systems," *ACM SIGSOFT Software Engineering Notes, Proceedings of the 8th European Software Engineering Conference*, vol. 26. no. 5, pp. 44-51, September 2001.

[22] C. Boyapati, R. Lee and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," *ACM SIGPLAN Notices, Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, vol. 37, no. 11, pp. 211-230, November 2002.

[23] G. Reeves, "What really happened on mars," in *RISKS Forum*, vol. 19, Issue 54, January 1998.

[24] E. McCluskey, "Minimization of Boolean Functions," in *Bell system technical Journal*, vol. 35, pp. 1417-1444, 1959.