A Parallel Deadlock Detection Algorithm with O(1) Overall Run-time Complexity

Jaehwan John Lee and Xiang Xiao
ECE Department, Purdue School of Engineering and Technology
Indiana University-Purdue University Indianapolis

Abstract

This article proposes a novel parallel, hardware-oriented deadlock detection algorithm for multiprocessor system-on-chips. The proposed algorithm takes full advantage of hardware parallelism in computation and maintains information needed by deadlock detection through classifying all resource allocation events and performing class specific operations, which together make the overall run-time complexity of the new method O(1). We implement the proposed algorithm in Verilog HDL and demonstrate in the simulation that each algorithm invocation takes at most four clock cycles in hardware.

I. Introduction

Embedded applications are becoming increasingly complicated nowadays. To meet the performance goals of current and future embedded applications, Multiprocessor System-on-Chips (MPSoCs) have emerged recently, which may combine many embedded processors, specialized digital logic and many hardware resources. While MPSoCs enable powerful embedded systems, the probability of a system entering a deadlock state is greatly increased since on an MPSoC many processes are likely to compete for shared resources. Furthermore, as MPSoCs are mostly designed for real-time embedded applications, they are managed by a Real-Time Operating System (RTOS). Because the users of an RTOS desire predictable response time, an RTOS must provide services (e.g., deadlock detection) that guarantee deadlines to be met deterministically. Thus, any deadlock detection approach proposed for MPSoCs must operate in a deterministic manner.

To meet the challenges in designing fast and deterministic deadlock detection methods for current and future MPSoCs, this article presents a parallel, hardware-oriented deadlock detection algorithm. When implemented in hardware, the algorithm has overall run-time complexity of $\mathcal{O}(1)$, which is the first in history to the best of our knowledge.

II. Related Work

Many algorithms have been devised to detect deadlock in the past. Shoshani et al. [8] presented an $\mathcal{O}(m^2 \times n)$ deadlock detection algorithm, where m is the number of processes and n is the number of resources. Holt introduced an $\mathcal{O}(m \times n)$ algorithm in [1]. Both [8] and [1] leveraged Resource Allocation Graph (RAG) to perform deadlock detection. Leibfried described an algorithm for deadlock detection using an adjacency matrix representation and matrix multiplication, but its run-time complexity is $\mathcal{O}(m^3)$ [3]. Kim et al. [2] proposed an algorithm that exploits a tree-like view of RAG and thus detects deadlock in $\mathcal{O}(1)$ run-time. However, Kim's method requires $\mathcal{O}(m + n)$ time to update auxiliary information to prepare for next deadlock detection. Shiu et al. [7] proposed a parallel algorithm which uses an adjacency matrix representation and graph reduction to detect deadlock, and its run-time complexity is $\mathcal{O}(min(m, n))$ in hardware. Recently, our previous work [12] has improved Kim's method by reducing the run-time of detection preparation to $\mathcal{O}(min(m, n))$. The algorithm presented in this article further reduces the run-time of detection preparation to $\mathcal{O}(1)$ when it is parallelized.

III. Our Methodology

In this section, we first present the assumptions made for the target systems. Next, we describe a Resource Allocation Graph (RAG) and its matrix representation and then introduce a classification of all resource allocation events. Lastly, we elaborate on the new algorithm, which works based on the classification.

A. Assumptions

For the algorithm, we make the following assumptions for its target systems: (i) Each resource contains only one unit. Hence, a cycle in the RAG becomes a necessary and sufficient condition for deadlock [1]. (ii) A resource is granted to a process immediately if that resource is available. As a result, the entire system is always in an *expedient* state [1]. (iii) One process requests one resource unit at a time. Thus, a process is blocked as soon as it requests an unavailable resource.

B. Resource Allocation Graph (RAG) And Its Adjacency Matrix Representation

A RAG contains a set of nodes (V) and a set of directed edges (E). A RAG is a bipartite graph in two disjoint sets: one process set *P* and the other resource set *Q*. Furthermore, there can be only two types of edges. One type (request edge) is directed from a node (p_i) in *P* to a node (q_j) in *Q*, which represents that process p_i is blocked when requesting q_j that is not available. The other type (grant edge) is directed from a node (q_j) in *Q* to a node (p_i) in *P*, which represents that resource q_j is granted to process p_i. A *cycle* is a sequence of alternating nodes and edges (q_{j_1},p_{i_1}), ..., (p_{i_k},$q_{j_{k+1}}$), ..., (q_{j_s},p_{i_s}) and an additional edge (p_{i_s},q_{j_1}), where each edge enclosed by () is distinct. Because a cycle is a necessary and sufficient condition for deadlock for the target systems, we are interested in finding cycles in a RAG. To find a cycle at a specific system state, we need to identify a reachable *sink* process node for each resource node. A sink process node is an active (i.e., not blocked) process with no outgoing edges. Note that under our assumptions, at most one sink process node is reachable from each resource node. The reachable sink process node for resource q_j is denoted as $\gamma_s(q_j)$.

Since there exist only two types of edges in a RAG, we use two matrices to represent all edges in a RAG. One matrix represents request edges, which we call Adjacency-Request matrix (denoted as *AR*). The other represents grant edges, which we call Adjacency-Grant matrix (denoted as *AG*). As a summary, $AR = \{AR[i][j]\}_{m\times n}$, $(1 \leq i \leq m, 1 \leq j \leq n)$, and $AG = \{AG[j][i]\}_{n\times m}$, $(1 \leq j \leq n, 1 \leq i \leq m)$, where m is the number of processes and n is the number of resources, and where

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

There are some particular properties in the two matrices. If process p_i is a sink node in a RAG, the ith row in *AR* contains all zeros. Furthermore, according to assumption (iii), each row in *AR* contains at most one 1. By assumption (i), each row in *AG* contains at most one 1.

C. A Classification of Resource Events

Critical to our algorithm is a classification of all possible events relevant to resource allocation. We first categorize all events into three types, each of which is then further classified to cover all possible scenarios. The three types of events are *granted resource request*, *blocked resource*

request and *resource release*. Note that we form each class if all scenarios in the class require the same operations for updating information on reachable sink node.

Let us look at *granted resource requests* first. For a resource (say q_j) to be assigned to a process (say p_i), q_j has to be available and p_i be an active process before the resource grant occurs. Because q_j contains only one unit and the system is in an expedient state, node q_j in the RAG can have neither incoming edges nor outgoing edges. Moreover, node p_i cannot have outgoing edges, either, but may have incoming edges. Since only q_j changes its reachable sink, there is only one class of grant scenarios, shown in Figure 1.

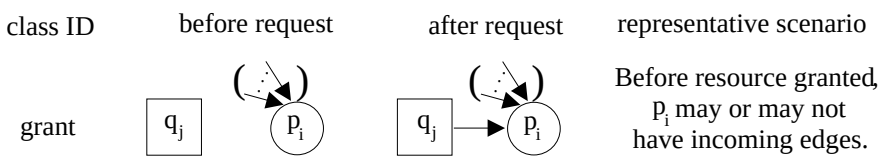


Fig. 1. One class of scenarios of *granted resource requests*.

Next, let us look at *blocked resource requests*. When process p_i is blocked for requesting resource q_j, p_i has to be an active process before requesting q_j. Thus, node p_i does not have outgoing edges before requesting q_j. However, because p_i may have obtained other resources, node p_i may have incoming edges. Furthermore, since resource q_j is not available at the time when p_i requests q_j, node q_j has one outgoing edge. Because there may already be other processes blocked for requesting q_j, node q_j may have incoming edges before the request. Taking these circumstances into account, two classes are formed, as illustrated in Figure 2.

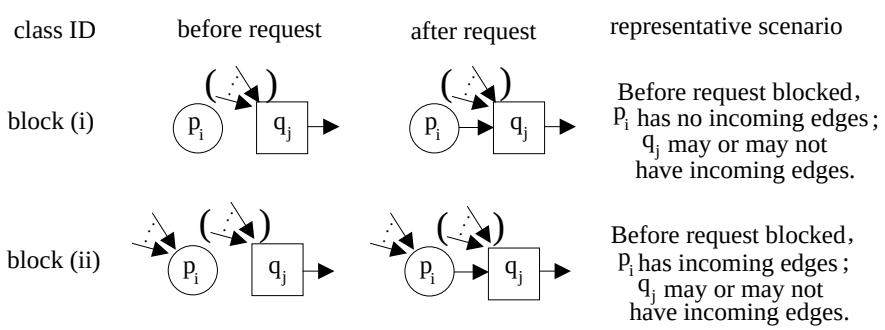


Fig. 2. Two classes of scenarios of *blocked resource requests*.

Lastly, let us look at *resource release* events. Only an active process (say p_i) can release one of its possessed resources (say q_j) once it no longer needs q_j. Hence, p_i has no outgoing edges. For

released resource q_j, node q_j certainly has an outgoing edge to p_i before the release. Depending on whether or not q_j has incoming edges (i.e., whether or not some processes wait for q_j) before the release, two different sets of operations for updating information on reachable sink nodes are needed. Hence, two classes for resource release events are formed, as depicted in Figure 3. In either class, p_i may have only one incoming edge from q_j before the release, or more than one incoming edges. In class release (i), q_j does not have any incoming edges. In class release (ii), q_j may have one or more incoming edges (from processes waiting for q_j). Moreover, in class release (ii), the process (say p_t) that q_j is granted to after the release may or may not have incoming edges. However, all these scenarios can be dealt with the same operations required for updating information on reachable sink nodes.

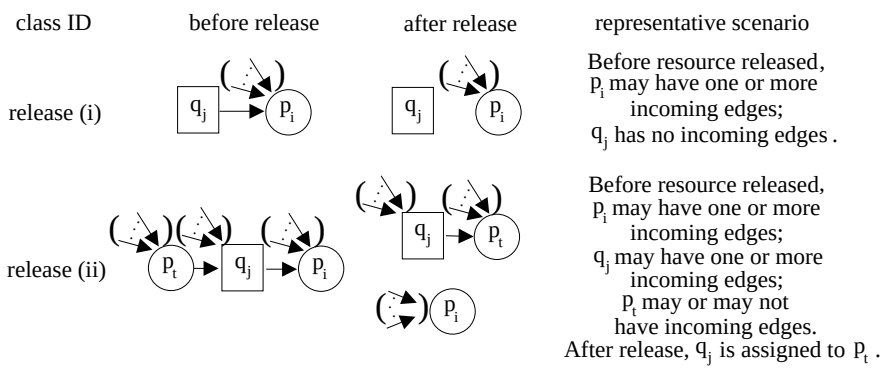


Fig. 3. Two classes of scenarios of *resource release* events.

Immediately after each event, the algorithm first determines which aforementioned class an event belongs to, and then performs a set of operations for the specific class described in Table I to update information on reachable sink nodes.

TABLE I

OPERATIONS FOR EACH CLASS OF RESOURCE ALLOCATION SCENARIOS.

class of scenarios	operations w.r.t. any change in sink nodes
grant	$\gamma_s(q_j) = p_i$; no change for other resources.
block (i)	no change in reachable sink nodes.
block (ii)	resources on p_i's sub-tree before the request change their reachable sink nodes to $\gamma_s(q_j)$.
release (i)	q_j now has no reachable sink node; no change for other resources.
release (ii)	after the release, only resources on q_j's subtree (including q_j) change their reachable sink nodes to p_t.

D. A New O(1) Parallel Deadlock Detection Algorithm

It is observed that as long as we know, for each resource node, which sink node can be reached via a path starting from the resource node, deadlock can be detected in $\mathcal{O}(1)$ time [2], [12]. Note that according to Assumption 2, only a process can be a sink. Thus, we use a sink node to refer to a sink process node. The reachable sink node of a resource (say q_j) is p_i if and only if p_i is a sink node and there exists a path from q_j to p_i. If the reachable sink node for a resource (say q_j) is known to be p_i, we know that a cycle forms when p_i requests q_j [2], [12]. The same idea to find cycles in a RAG was adopted in Kim's algorithm [2] and our previous work [12]. However, with this deadlock detection approach, the reachable sink node of each resource has to be re-computed every time a new resource event occurs so that the new information will be used for next deadlock detection invocation. The re-computation of reachable sink nodes has run-time complexity of $\mathcal{O}(m+n)$ in [2] and $\mathcal{O}(min(m,n))$ in [12]. This overhead makes the overall run-time complexity of either of the previous algorithms not $\mathcal{O}(1)$ even though the detection only takes $\mathcal{O}(1)$ time. On the contrary, the algorithm presented in this article is able to make the run-time complexity of the re-computation of reachable sink nodes $\mathcal{O}(1)$ as well. Hence, the run-time complexity of the entire algorithm is truly $\mathcal{O}(1)$.

To implement the operations described in Table I, the proposed algorithm utilizes a matrix called Sink to keep information on the reachable sink nodes of all resources. Moreover, for class release (ii), the algorithm needs to identify resources on the sub-tree of the released resource (say q_j) as well as those on the sub-tree of the process newly acquiring q_j. Hence, we use matrices ReachableResource (or RR in short) and ReachableProcess (or RP in short) to keep information on what resources and processes are reachable from every resource, respectively. These three matrices are defined as follows.

$$Sink[j][i]_{n\times m} = \begin{cases} 1 & \text{if } p_i \text{ is the reachable sink node of } q_j, \\ 0 & \text{otherwise,} \end{cases}$$

$$RR[j][k]_{n\times n} = \begin{cases} 1 & \text{if a path exists from resource } q_j \text{ to } q_k \text{ or k == j,} \\ 0 & \text{otherwise, and} \end{cases}$$

$$RP[j][i]_{n\times m} = \begin{cases} 1 & \text{if a path exists from resource } q_j \text{ to } p_i, \\ 0 & \text{otherwise.} \end{cases}$$

Before we present the pseudo code of the algorithm, let us first introduce some notations. In this article, matrix[], matrix[i][] and matrix[][j] refer to as "all elements in the matrix," "all elements of row i in the matrix," and "all elements of column j in the matrix," respectively. Next, the pseudo code of our parallel deadlock detection algorithm is presented in Algorithm 1. Explanation of the algorithm in detail is given in the next subsection.

ALGORITHM 1 *O(1) Parallel Deadlock Detection Algorithm*

Part 1: Resource request granted (p_i, q_j)

$AG[j][i] \leftarrow 1$ // q_j is granted to p_i.
$Sink[j][i] \leftarrow 1$ // The reachable sink node of q_j is now p_i.
$RP[j][i] \leftarrow 1$ // p_i also becomes a reachable process of q_j.
return

Part 2: Resource request blocked (p_i, q_j)

if $Sink[j][i] = 1$ // if q_j's reachable sink node is p_i, deadlock occurs.
  then *deadlock*
else
  $AR[i][j] \leftarrow 1$ // p_i is now blocked for requesting q_j.
  if *AG[][i] not all zeros* // if p_i possesses resources,
    // this event belongs to *block (ii)*.
    for all k where $k \leftarrow 1$ to n // in parallel for all resources ($O(1)$)
      do if $Sink[k][i] = 1$ // For all the resources
        // that belong to the sub-tree of p_i, their sink node are now set
        // to q_j's; their corresponding rows in RR and RP include q_j's.
        then $Sink[k][] \leftarrow Sink[j][]$ // parallel bitwise row copy
          // in hardware in $O(1)$ (multi-level parallelism)
          $RR[k][]^{new} \leftarrow RR[k][]^{old} \vee RR[j][]$
          $RP[k][]^{new} \leftarrow RP[k][]^{old} \vee RP[j][]$
        // parallel bitwise OR in hardware (multi-level parallelism)
return

Part 3: Resource release (p_i, q_j, p_t)

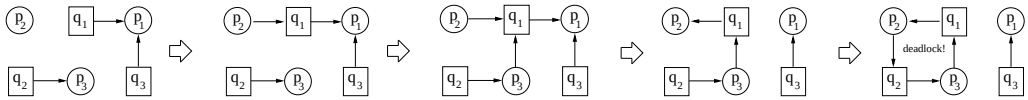
$AG[j][i] \leftarrow 0$ // p_i has released q_j.
if $AR[][j]$ *contains all zeros* // if no process is waiting for q_j,
  // this event belongs to *release (i)*.
  then $Sink[j][i] \leftarrow 0$ // q_j is isolated;
    // thus q_j does not have a sink node.
    $RP[j][i] \leftarrow 0$ // p_i is not a q_j's reachable process node, either.
else // Any process is waiting for q_j; this event belongs to *release (ii)*.

$AG[j][t] \leftarrow 1$; $AR[t][j] \leftarrow 0$ // q_j is now granted to p_t.
$NewSink[]_{1\times m} \leftarrow [000\ldots1\ldots00]$ *where* t^{th} *element is* 1
// representing new sink p_t.
for all k where $k \leftarrow 1$ to n // in parallel for all resources ($O(1)$)
  do if $RR[k][j] = 1$ // For q_j and its sub-tree resources,
    then $Sink[k][] \leftarrow NewSink[]$; $RP[k][i] \leftarrow 0$
  // their sink nodes are now set to p_t and p_i is no longer reachable.
      if $RP[k][t] = 1$ // For p_t's sub-tree resources that were able
      // to reach q_j, q_j is no longer reachable after the release.
        $RR[k][j] \leftarrow 0$
      else // For q_j's sub-tree resources that were not reachable
      // to p_t, p_t becomes reachable after the release.
        $RP[k][t] \leftarrow 1$
return

*$\vee$ denotes bitwise OR logical operation.

E. Explanation of the Algorithm with a Simple Example

To explain in detail how the algorithm works, we introduce the following example. The example system consists of three processes, denoted by p_1 through p_3, and three resources with each having one unit, denoted by q_1 through q_3. The RAGs of the example system at different moments are shown in Figure 4. Each event (i.e., granted resource request, blocked resource request or resource release) changes the RAG from one moment to the next moment (e.g., t_0 to t_1; t_2 to t_3). At each new moment, Algorithm 1 is invoked. In the following, we will show what event occurs, how Algorithm 1 is used for deadlock detection at moment t_1 to t_4 and how reachable sink node information is computed at each moment.



(a) at moment t_0 (b) at moment t_1 (c) at moment t_2 (d) at moment t_3 (e) at moment t_4

Fig. 4. A sequence of transitions of resource allocation in the example system. Note that $t_0 < t_1 < t_2 < t_3 < t_4$.

We assume that Figure 4(a) shows the initial state of the system at moment t_0. The values of AR, AG, $Sink$ and RR as well as RP at moment t_0 are listed in Figure 5. $Sink$ signifies that both q_1's and q_3's sink nodes are p_1, and q_2's sink node is p_3. RP indicates that each resource can only reach the process to which this resource is granted (i.e. $RP = AG$).

	q_1	q_2	q_3
p_1	0	0	0
p_2	0	0	0
p_3	0	0	0

(a) AR

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(b) AG

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(c) Sink

	q_1	q_2	q_3
q_1	1	0	0
q_2	0	1	0
q_3	0	0	1

(d) RR

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(e) RP

Fig. 5. Matrix values at moment t_0.

Then, at moment t_1, p_2 requests q_1 that is not available and thus p_2 is blocked (see Figure 4(b)). Because the request is blocked, *Part 2* of Algorithm 1 is invoked. In line 5, $Sink[1][2]^{t_0}$ is checked first. Since $Sink[1][2]^{t_0}$ is 0, this request does not cause deadlock. This decision takes $\mathcal{O}(1)$ run-time. Then, in line 8, we update $AR[2][1]$ to 1 to record this new blocked request. Since p_2 has been granted no resources (i.e. $AG[][2]^{t_0}$ = $[000]^T$), this event belongs to class *block (i)* (see Figure 2). As an effect, the *if* condition in line 9 becomes false, and thus matrices other than AR do not change. The algorithm finishes its computation on this event. The values of the matrices after moment t_1 are listed in Figure 6.

	q_1	q_2	q_3
p_1	0	0	0
p_2	1	0	0
p_3	0	0	0

(a) AR

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(b) AG

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(c) Sink

	q_1	q_2	q_3
q_1	1	0	0
q_2	0	1	0
q_3	0	0	1

(d) RR

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(e) RP

Fig. 6. Matrix values at moment t_1.

At moment t_2, p_3 requests q_1 and is blocked (see Figure 4(c)). Thus, *Part 2* of Algorithm 1 is invoked. No deadlock is detected because $Sink[1][3]^{t_1}$ is 0. Since p_3 has an incoming edge from q_2 (i.e., $AG[2][3]^{t_1} = 1$), this event belongs to class *block (ii)* (see Figure 2). Thus, the *if* condition in line 9 becomes true, and lines 10 to 14 are executed. Because $Sink[][3]^{t_1}$ is $[010]^T$, only the case of $k = 2$ matters among all k's. Thus, in lines 12 to 14, $k = 2$ as well as $j = 1$. Hence, firstly the 1st row of $Sink$ is copied into the 2nd row of $Sink$. In hardware, copying all elements of a row can be performed simultaneously so that the copy takes $O(1)$ time. Secondly, the 2nd row of RR (RP) is combined with the 1st row of RR (RP) using bitwise OR operation, and the 2nd row is updated with the result. Once again, the OR operations between all pairs of elements are performed simultaneously in hardware, which results in $O(1)$ time. The new values of the matrices are listed in Figure 7.

	q_1	q_2	q_3
p_1	0	0	0
p_2	1	0	0
p_3	1	0	0

(a) AR

	p_1	p_2	p_3
q_1	1	0	0
q_2	0	0	1
q_3	1	0	0

(b) AG

	p_1	p_2	p_3
q_1	1	0	0
q_2	1	0	0
q_3	1	0	0

(c) Sink

	q_1	q_2	q_3
q_1	1	0	0
q_2	1	1	0
q_3	0	0	1

(d) RR

	p_1	p_2	p_3
q_1	1	0	0
q_2	1	0	1
q_3	1	0	0

(e) RP

Fig. 7. Matrix values at moment t_2.

At moment t_3, p_1 releases q_1, which is granted to p_2 (see Figure 4(d)). *Part 3* of Algorithm 1 is invoked where $i = 1$, $j = 1$ and $t = 2$. Because q_1 is no longer granted to p_1, $AG[1][1]^{t_3}$ is assigned with 0. According to our classification, this event belongs to class *release (ii)* (see Figure 3). In line 17, since $AR[][1]^{t_2}$ ($[011]^T$) contains two ones, lines 21 to 29 are executed. In line 21, $AG[1][2]^{t_3}$ is assigned with 1 and $AR[2][1]^{t_3}$ with 0 because q_1 is now granted to p_2. After this event, every resource node whose reachable resource nodes includes q_1 before the release is no longer reachable to p_1 and needs to change its reachable sink node to p_2 (line 25). Parallelized bitwise assignment is leveraged to make this multi-element assignment $O(1)$ time. In addition, before the release, q_1 and q_2 were on q_1's sub-tree but not p_2's (i.e., skipping lines 26 and 27, but executing lines 28 and 29). Thus, $RP[1][2]^{t_3}$ and $RP[2][2]^{t_3}$ now become ones, indicating p_2 also accounts for a reachable process node of q_1 as well as q_2. The new values of the matrices are listed in Figure 8.

	q_1	q_2	q_3
p_1	0	0	0
p_2	0	0	0
p_3	1	0	0

(a) AR

	p_1	p_2	p_3
q_1	0	1	0
q_2	0	0	1
q_3	1	0	0

(b) AG

	p_1	p_2	p_3
q_1	0	1	0
q_2	0	1	0
q_3	1	0	0

(c) Sink

	q_1	q_2	q_3
q_1	1	0	0
q_2	1	1	0
q_3	0	0	1

(d) RR

	p_1	p_2	p_3
q_1	0	1	0
q_2	0	1	1
q_3	1	0	0

(e) RP

Fig. 8. Matrix values at moment t_3.

At moment t_4, process p_2 requests resource q_2 and is blocked (see Figure 4(e)). *Part 2* of Algorithm 1 is invoked. Because $Sink[2][2]^{t_3}$ is 1, which signifies that p_2 is q_2's reachable sink node before the request, Algorithm 1 detects deadlock in line 6. It is because p_2's request for q_2 forms a cycle in the RAG. Equivalently, deadlock occurs in the system.

Through this example, it can be seen that bitwise matrix operations for each of matrices $Sink$, RR and RP require a separate set of $\mathcal{O}(n)$ processing units, where n is the number of resources. In such a way, not only the detection but also the computation of reachable sink nodes can be completed in $\mathcal{O}(1)$ time in hardware. Proofs of the correctness and run-time complexity of Algorithm 1 are presented in [11].

IV. Experimentation and Results

A. Target System and Simulation Environment

To demonstrate the correctness and run-time of the proposed algorithm, we implement it in Verilog HDL, which we call O(1) Hardware Deadlock Detection Unit (OHDDU), and integrate it

with a simulated multiprocessor computer system. The multiprocessor system contains five Motorola MPC755 processors, on which five processes ($p_1, \ldots, p_5$) run (one process per processor for simplicity). It also has SRAM memory of 256MB and four dummy resources ($q_1, \ldots, q_4$). These dummy devices count the specified time (set by the software) after they are granted and interrupt the corresponding processor where a process uses the resource when a preset time elapses. Thus, it is sufficient to use these dummy resources to simulate various deadlock scenarios in the system. All components, except the processors and SRAMs, are implemented in Verilog HDL. The application is written in the C language and compiled using a PowerPC-GCC cross-compiler. We use Atalanta RTOS version 0.3 [9], a shared memory multiprocessor RTOS, to manage processes and resources. In order to simulate such multiprocessor architecture with the multiprocessor operating system, we use Mentor Graphics Seamless [4] aided by ModelSim for hardware simulation and XRAY for software debugging. We also use the Mentor Graphics ASIC Design Kit 3.0 [5] with TSMC .25 μm standard cell library [10] to synthesize a 5x4 (i.e., for 5 processes and 4 resources) OHDDU. The area of a synthesized 5x4 OHDDU is 1096 units, each of which is equivalent to a minimum-sized two-input NAND gate in the library.

B. Simulation and Its Results

We simulated a series of resource allocation events listed in Table II, which covers all three types of resource events (simulation of more complicated examples is presented in [11]). First, we measured how many clock cycles the algorithm takes to conclude whether or not deadlock occurs when a process is blocked for a request (lines 5 to 6 of Algorithm 1). Then, we measured how many clock cycles the algorithm takes to compute the reachable sink nodes for all resources after each event. Our simulation shows that deadlock detection, if needed, takes always 2 cycles, and the computation of reachable sink nodes for all resources takes at most 2 cycles. In cases of events 4 and 9, because the algorithm only needs to update matrix AR but no others, the computation takes 1 cycle for such events. In case of event 11, deadlock is detected in 2 cycles. After event 11, we assume that some deadlock recovery scheme is carried out, and thus, there is no immediate re-computation of reachable sink nodes.

V. Conclusion

A parallel, hardware-oriented deadlock detection algorithm suitable for Multiprocessor System-on-Chips (MPSoCs) is presented in this article. During the phase of detection preparation, the

TABLE II

SIMULATED RESOURCE EVENTS IN THE TIMING ORDER.

sequence	events	detection clock cycles	preparation clock cycles
1	p_1 requests q_1, q_1 granted to p_1	NA	2
2	p_3 requests q_2, q_2 granted to p_3	NA	2
3	p_1 requests q_3, q_3 granted to p_1	NA	2
4	p_2 requests q_1, blocked	2 (no deadlock)	1
5	p_3 requests q_1, blocked	2 (no deadlock)	2
6	p_1 releases q_1 and q_1 granted to p_2	NA	2
7	p_5 requests q_4, q_4 granted to p_5	NA	2
8	p_5 releases q_4	NA	2
9	p_5 requests q_3, blocked	2 (no deadlock)	1
10	p_1 requests q_2, blocked	2 (no deadlock)	2
11	p_2 requests q_3, blocked	2 (deadlock)	NA

presented algorithm handles different types of resource allocation events differently based on a classification of all resource allocation events. Furthermore, taking full advantage of hardware parallelism, the new algorithm is able to complete preparation operations for every type of event in $\mathcal{O}(1)$ run-time in hardware. Since both detection preparation and detection itself take $\mathcal{O}(1)$ run-time, the overall run-time complexity of the presented algorithm is $\mathcal{O}(1)$. The algorithm can be easily implemented as a specialized function unit for MPSoCs with small area overhead.

REFERENCES

[1] R. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, no. 3, 1972, pp. 179–196.

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

[3] T. Leibfried, "A deadlock detection and recovery algorithm using the formalism of a directed graph matrix," *Operating Systems Review*, vol. 23, no. 2, 1989, pp. 45–55.

[4] Mentor Graphics, Hardware/Software Co-Verification: Seamless, http://www.mentor.com/seamless, visited March 2008.

[5] Mentor Graphics, ASIC Design Kit. http://www.mentor.com/company/higher_ed/asic/index.cfm, visited March 2008.

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

[7] P. Shiu, Y. Tan and V. Mooney, "A novel parallel deadlock detection algorithm and architecture," *Proc. CODES 01*, 2001, pp. 73–78.

[8] A. Shoshani and J. E. Coffman, "Detection, prevention and recovery from deadlocks in multiprocess, multiple resource systems," Tech. Report, 80, Princeton University, 1969.

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

[10] TSMC, Taiwan Semiconductor Manufacturing Co. Ltd. http://www.tsmc.com/, visited March 2008.

[11] X. Xiao and J. Lee, "A parallel deadlock detection algorithm with O(1) overall run-time complexity and its hardware implementation," Tech. Report, TR-ECE-08-02, ECE, IUPUI, 2008.

[12] X. Xiao and J. Lee, "A novel parallel deadlock detection algorithm and hardware for multiprocessor system-on-a-chip," *IEEE Computer Architecture Letters*, vol. 6, no. 2, Jul-Dec, 2007, pp. 41–44.