A Novel Deadlock Avoidance Algorithm and Its Hardware Implementation

Jaehwan Lee
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
jaehwan@ece.gatech.edu

Vincent John Mooney III
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
mooney@ece.gatech.edu

ABSTRACT

This paper proposes a novel Deadlock Avoidance Algorithm (DAA) and its hardware implementation, the Deadlock Avoidance Unit (DAU), as an Intellectual Property (IP) core that provides a mechanism for very fast and automatic deadlock avoidance in MultiProcessor System-on-a-Chip (MPSoC) with multiple (e.g., 10) processing elements and multiple (e.g., 40) resources. The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock. In case of livelock, the DAU asks one of the processes involved in the livelock to release resource(s) so that the livelock can also be resolved. We simulated two realistic examples that can benefit from the DAU, and demonstrated that the DAU not only avoids deadlock in a few clock cycles but also achieves a 37% speed-up of application execution time over avoiding deadlock in software. Finally, the SoC area overhead due to the DAU is small, under 0.01% in our example.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time and Embedded Systems–Hardware Support; D.4.1 [Software]: Operating Systems–Deadlock

General Terms: Algorithms, Design, Experimentation

Keywords: Deadlock Avoidance Hardware IP design

1. INTRODUCTION AND MOTIVATION

In most current embedded systems, deadlock is not a critical issue due to the use of only a few (e.g., two) processors and a couple of custom hardware resources (e.g., direct memory access hardware plus a video decoder). However, in the coming years future chips may have five to twenty processors and ten to a hundred resources all in a single chip as shown in Figure 1. This is the way we predict MultiProcessor System-on-a-Chip (MPSoC) will rapidly evolve. Even in the platform design area, Xilinx already has been able to include multiple PowerPC processors in the Virtex-II Pro FPGA [1]. Given current technology trends, we predict that MPSoC designers and users are going to start facing deadlock problems more and more often.

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

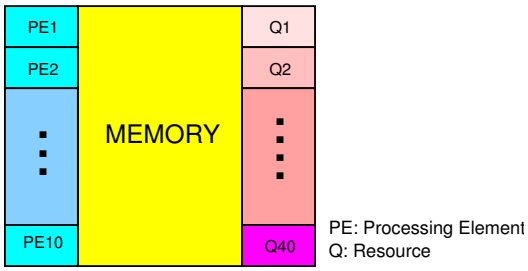

Figure 1: Future MPSoC.

How can we efficiently and timely cope with deadlock problems in such an MPSoC? Although MPSoC may produce deadlock problems, MPSoC architecture can also provide efficient hardware solutions to deadlock. This paper proposes one such solution, a novel Deadlock Avoidance Algorithm (DAA) and its hardware implementation, the Deadlock Avoidance Unit (DAU), to improve the reliability and correctness of applications running on an MPSoC under an RTOS. Of course, adding a centralized module on MPSoC may lead to bottleneck. However, since resource allocation and deallocation are preferably managed by an operating system (which already implies some level of centralized operation), adding hardware can potentially reduce the burden on software rather than becoming a bottleneck.

Traditional deadlock avoidance essentially requires a priori knowledge about the maximum necessary resource requirements for all processes in a system [2, 3], which unfortunately makes the implementation of deadlock avoidance difficult in real systems. Our novel approach to mixing deadlock detection and avoidance (thus, not requiring advanced, a priori knowledge of resource requirements) contributes to easier adaptation of deadlock avoidance in an MPSoC by accommodating both maximum freedom (i.e., maximum concurrency of requests and grants depending on a particular execution trace) with the advantage of deadlock avoidance.

From prior work we utilize a novel parallel deadlock detection algorithm (PDDA) and a deadlock detection hardware unit (DDU) [4] that has a computational complexity of $O(min(m,n))$ [5]. The DDU has a matrix mapped from a resource allocation graph (RAG). Request and grant edges of a RAG are translated into elements of a matrix, each element of which represents (stores) either request-, grant-, or no-edge. During deadlock detection, reducible edges that are not involved in deadlock are temporarily removed from the matrix iteratively. If all edges are reducible, there exists no deadlock; otherwise, there exists a deadlock. The DDU traces neither cycles nor paths, nor requires linked lists. Not only that, because a series of detection operations can occur in one clock cycle throughout rows and columns of a ma-

trix in parallel hardware, the detection has been proven to take at most $(2 \times min(m, n) - 3)$ cycles [5]. Much detailed information about the DDU is described in [5]. Unlike the DDU [4, 5], we have thought that it would be very helpful if there were a hardware unit that not only detects deadlock but also avoids possible deadlock within a few clock cycles and with a small amount of hardware. In Section 6 we will describe scenarios where our approach reduces deadlock avoidance time by up to 312X resulting in application speedup of up to 44%, all at a cost of only 0.01% increase in SoC area.

2. PREVIOUS WORK

A traditional well-known deadlock avoidance algorithm is the Banker's algorithm [3]. The algorithm requires each process to declare the maximum requirement (claim) of each resource it will ever need. In general, deadlock avoidance is more expensive than deadlock detection or may be impractical because of the following disadvantages: (i) an avoidance algorithm must be executed for every request prior to granting a resource, (ii) deadlock avoidance restricts resource utilization, which degrades system performance, and (iii) the maximum resource requirements (and thus requests) could not be known in advance [2, 3]. In contrast, the DAU requires neither prior knowledge about requirements of processes nor constraints of resource usage, yet achieves real-time deadlock avoidance; this constitutes the major novelty in our solution to deadlock avoidance.

In 1990, Belik proposed a deadlock avoidance technique [6], in which a path matrix representation is used to detect a potential deadlock before the actual allocation of resources. However, Belik's method requires $O(m \times n)$ time complexity for updating the path matrix in releasing or allocating a resource and thus an overall complexity for avoiding deadlock of $O(m \times n)$, where m and n are the numbers of resources and processes, respectively. Furthermore, Belik does not mention any solution to livelock although livelock is a possible consequence of deadlock avoidance.

Although many deadlock avoidance approaches have been introduced so far [2, 3, 6, 7, 8], to the best of our knowledge, there has been no prior work in a hardware implementation of deadlock avoidance. The reason we conjecture is that MPSoC is not full-blown yet; thus people tend to neither notice nor pay attention to coming deadlock problems, which we predict are on the horizon. The DAU we present not only provides a solution to both deadlock and livelock but is also up to 312X faster than an equivalent software solution (please see the details in Section 6).

3. DEFINITIONS AND SYSTEM MODEL

In this section, we first mention some deadlock definitions for better understanding our methodology to be introduced in Section 4. Then, we describe our system model.

3.1 Definitions

Definitions of deadlock and livelock in our context can be stated as follows.

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

DEFINITION 2. *Livelock is a situation where a request for a resource is repeatedly denied because of the unavailability of the resource.*

In addition, we define two kinds of deadlock: request deadlock (R-dl) and grant deadlock (G-dl).

DEFINITION 3. *For a given system, if a request from a process causes the system to have a deadlock, then we denote this case as* ***request deadlock*** *or* ***R-dl****.*

EXAMPLE 1. Request deadlock (R-dl) Example
In an MPSoC application, on-chip processors may have to use several resources, for example, to process streaming data. Figure 2 shows such a system having two processors, a Very-Long Instruction Word (VLIW) Processor (VP) and a Specialized Processor (SP), and two resources, a Bluetooth Interface (BI) and a Moving Picture Experts Group (MPEG) decoder. Each processor (VP or SP) has to use both resources exclusively to complete its processing of the streaming data. In the case shown in Figure 2(b), VP holds resource MPEG while SP holds resource BI. (Please see the event sequence marked on the side of each edge shown in Figure 2(b).) Furthermore, VP requests BI, and SP requests MPEG. When SP requests MPEG, the system will have a deadlock since neither VP nor SP gives up or releases the resources they currently hold; instead, they wait for their requests to be fulfilled. We denote this case, in which a request causes a deadlock, as **request deadlock** or **R-dl**. □

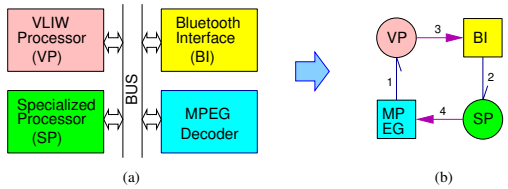

Figure 2: Request deadlock (R-dl) example.

DEFINITION 4. *For a given system, if the grant of a resource to a process causes the system to have a deadlock, then we denote this case as* ***grant deadlock*** *or* ***G-dl****.*

EXAMPLE 2. Grant deadlock (G-dl) Example
We show a sequence of requests and grants that leads to a deadlock as shown in Figure 3. It is assumed that p_2 has a priority higher than p_3. At time t_1, process p_1 requests both q_1 and q_2, which are then granted to p_1. After that, p_1 starts working. At time t_2, p_3 requests q_2 and q_3. However, only q_3 is granted to p_3 since q_2 is unavailable. At time t_3, p_2 also requests q_2 and q_3, which are not available for p_2 yet. When the computation of p_1 is done, q_1 and q_2 are released by p_1 at time t_4. Then q_2 is granted to p_2 at time t_5 since p_2 has a priority higher than p_3. This last grant will lead to a deadlock in the system, which we denote as **grant deadlock** or **G-dl**. □

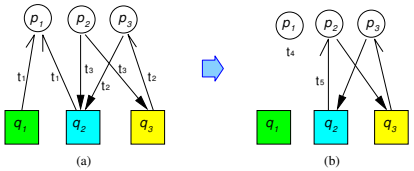

Figure 3: Grant deadlock (G-dl) example.

3.2 Our System Model

To describe our system model, we first show a possible MPSoC in the following example.

EXAMPLE 3. A future Request-Grant MPSoC
We introduce the device shown in Figure 4 as a particular MPSoC example. This MPSoC consists of four Processing Elements (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, as shown in Figure 4(b).

The MPSoC also contains memory, a memory controller and a DAU. 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 4(b), runs on PE1, PE2, PE3 and PE4, respectively. In the current state, 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. The DAU in Figure 4 receives all requests and releases, decides whether or not the request or grant can cause a deadlock and then permits the request or grant only if no deadlock results. □

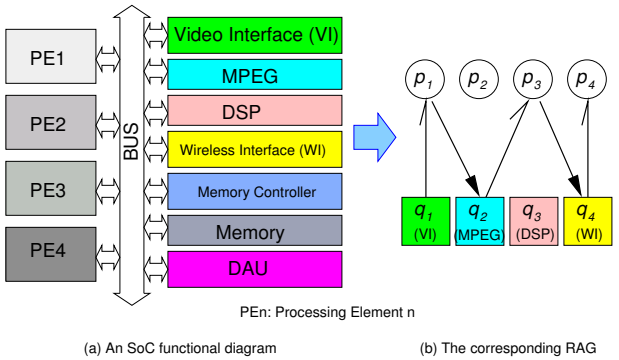

Figure 4: A practical MPSoC realization.

We consider this kind of request-grant system with many resources and processes shown in Figure 4 as our system model. Based on our system model, we now introduce some underlying assumptions related to our deadlock avoidance research in such MPSoCs.

ASSUMPTION 1. *In our system model, there exists a fixed number of resources.*

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.*

Note that Assumption 2 disallows multiple-unit or pipelined resources. As future work, we intend to relax Assumption 2 and thus support multiple-unit and pipelined resources.

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

ASSUMPTION 4. *The RTOS provides a mechanism that can ask a process to release any resource(s) the process currently holds.*

ASSUMPTION 5. *In our system model, all processes have unique priorities.*

Note that for our future work, we will support systems with non-priority based scheduling as well, e.g., round-robin.

4. METHODOLOGY

4.1 Our Deadlock Avoidance Method

This section introduces the main concept of the DAU. The DAU, if employed, tracks all requests and releases of resources. In other words, the DAU receives, interprets and executes commands from processes; then it returns DAU processing results back to processes. The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock.

Algorithm 1 shows our first approach to deadlock avoidance. When a process requests a resource from the DAU (line 2), the DAU checks for the availability of the resource requested (line 3). If the resource is available (i.e., no one is using it), the resource will be granted to the requester immediately (line 4). If the resource is not available, the DAU checks the possibility of request deadlock (R-dl) (line 5). If the request would cause R-dl, the DAU does not accept the request (i.e., the request is denied); thus R-dl can be avoided (line 6). On the other hand, if the request does not cause R-dl (line 7), the DAU makes the request be pending since the resource is not available (line 8).

When the DAU receives a resource release command from a process (line 11), if no process is waiting for the resource (line 18), the resource simply becomes available (line 19). On the other hand, if a process is waiting for it, the DAU checks for the possibility of grant deadlock (G-dl) (line 13) and next grants the resource to the requester only if the grant does not result in G-dl (line 16). If, however, the grant would cause G-dl, the resource is not granted (line 14).

ALGORITHM 1. Deadlock Avoidance Algorithm (DAA)

```
DAA (event) {
1   case (event) {
2       a request:
3           if the resource is available
4               grant the resource to the requester
5           else if the request would cause request deadlock (R-dl)
6               deny the request
7           else
8               make the request be pending
9           end-if
10          break;

11      a release:
12          if any process is waiting for the released resource
13              if the grant of the resource would cause grant deadlock
14                  do not grant the resource
15              else
16                  grant the resource to the process waiting
17              end-if
18          else
19              make the resource become available
20          end-if
21  } end-case
}
```

The above scheme will avoid deadlock. However, in line 6 of Algorithm 1, when a request is denied because of potential request deadlock (R-dl), the situation may introduce the starvation of the processes involved in the potential R-dl (i.e., even though a system does not have a deadlock, no progress can be made by some processes, which is also known as livelock). In addition, in line 14 of Algorithm 1, although a resource becomes available, it cannot be granted because of grant deadlock (G-dl), resulting in resource underutilization and/or livelock. Thus, the above scheme requires some modification.

Here, we propose two novel approaches. Algorithm 2 implements an approach that avoids not only deadlock but also livelock. When a request would cause R-dl (line 5), the request is denied with an error code telling the requester that it is potentially in R-dl (line 6, i.e., currently in livelock) by setting the R-dl bit in a status register the requester reads. In this way, the requester is informed of potential livelock; we assume that the requester voluntarily releases some resource(s) it holds in order to remove the possibility of livelock.

In addition, when the DAU receives a resource release command from a process (line 11) and any process is waiting for the resource (line 12), before actually granting the released resource to one of the requesters, the DAU temporarily marks a grant of the resource to the highest priority process (on its internal matrix). Then, to check potential grant deadlock, the DAU executes its deadlock detection algorithm. If the temporary grant does not cause

grant deadlock (G-dl) (line 15), it becomes a fixed grant; thus the resource is granted to the highest priority requester (line 16). On the other hand, if the temporary grant causes G-dl (line 13), the temporary grant will be undone; then, because the released resource cannot be granted to the highest priority requester because of G-dl, the DAU tries to grant the resource to a lower priority requester (line 14). The DAU continues checking all processes to see if the released resource can be granted to a process without the involvement of deadlock. As a result, resources can be effectively exploited. Other behaviors are the same as Algorithm 1.

ALGORITHM 2. DAA (Approach Two)

```
DAA (event) {
1   case (event) {
2       a request:
3           if the resource is available
4               grant the resource to the requester
5           else if the request would cause request deadlock (R-dl)
6               deny the request and inform the potential R-dl (i.e., livelock)
                (let the requester take care of this livelock situation)
7           else
8               make the request be pending
9           end-if
10          break;

11      a release:
12          if any process is waiting for the released resource
13              if the grant of the resource would cause grant deadlock
14                  grant the resource to a lower priority process waiting
15              else
16                  grant the resource to the highest priority process waiting
17              end-if
18          else
19              make the resource become available
20          end-if
21  } end-case
}
```

While the second approach is a good strategy, it is somewhat passive since the resolution of livelock solely depends on the last requester having caused the potential request deadlock (R-dl). Additionally, the request case of Algorithm 2 does not consider the importance (i.e., priorities) of processes competing for resources. Thus, in order to more actively and efficiently resolve livelock, we propose another approach.

ALGORITHM 3. DAA (Approach Three)

```
DAA (event) {
1   case (event) {
2       a request:
3           if the resource is available
4               grant the resource to the requester
5           else if the request would cause request deadlock (R-dl)
6               if the priority of the requester greater than that of the owner
7                   make the request be pending
8                   ask the current owner of the resource to release the resource
9               else
10                  ask the requester to give up resource(s)
11              end-if
12          else
13              make the request be pending
14          end-if
15          break

16      a release:
17          the same as Algorithm 2
18  } end-case
}
```

As shown in Algorithm 3, if a request would cause request deadlock (R-dl) (line 5) – note that the DAU tracks all requests and releases – the DAU compares the priority of the requester with that of the current owner of the requested resource. If the priority of the requester is higher than that of the current owner of the resource (line 6), the DAU makes the request be pending for the requester (line 7), and then the DAU asks the owner of the resource to give up the resource so that the higher priority process can proceed (line 8, the current owner may need time to finish or checkpoint its current processing). On the other hand, if the priority of the requester is lower than that of the owner of the resource (line 9), the DAU asks the requester to give up the resource(s) that the requester already has but is most likely not using yet (since all needed resources are not yet granted, line 10). Other behaviors are the same as Algorithm 2.

Either Algorithm 2 or Algorithm 3 can potentially be employed in a system. For instance, Algorithm 2 can be used in a system that does not satisfy Assumption 4. We chose to implement Algorithm 3 in hardware because it resolves livelock more actively and efficiently than Algorithm 2, in which the resolution of livelock depends on the last requester without considering priorities of processes.

4.2 Run-time Complexity of DAU

The DAU becomes active and starts working only when a request or a release event occurs. Once the DAU is activated, it operates in at most $2 \times min(m, n) + (n \times k)$ clock cycles, where m and n are the numbers of resources and processes, respectively, and where k is the number of cycles that each trial of unsuccessful grants takes, stated in line 14 of Algorithm 2. This run-time is a theoretical lower bound on the complexity and is valid as long as the clock period is longer enough than the maximum delay of a critical calculation. After finishing operation, the DAU remains idle until a next event occurs. Processes and the DAU communicate via specific application programming interfaces (APIs).

5. IMPLEMENTATION

5.1 Architecture of the DAU

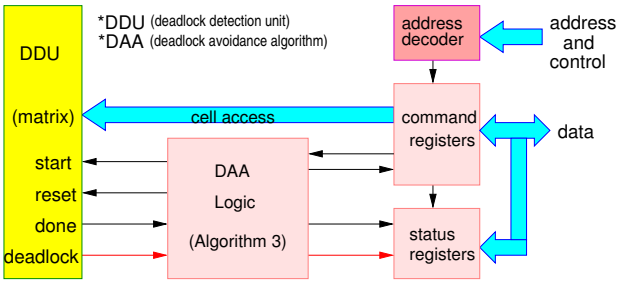

Figure 5: DAU architecture.

Figure 5 illustrates the DAU, implemented in the Verilog Hardware Description Language (HDL). The DAU consists of four parts: a Deadlock Detection Unit (DDU [4]), command registers, status registers and a unit implementing Algorithm 3 with a finite state machine. The DDU employs a matrix of requests and grants for quickly detecting deadlocks. Command registers receive commands from each PE. The processing results of the DAU are stored into status registers read by all PEs. While a command register contains a release or request of a resource, a status register contains the information of *done, busy, successful, pending, give-up, which-process, which-resource, livelock* as well as *G-dl* and *R-dl*. The DAA logic mainly controls the DAU behavior, i.e., interprets and executes commands (requests or releases) from PEs, and returns processing results back to PEs via status registers.

5.2 Synthesized Result of the DAU

We used the Synopsys Design Compiler (DC) [12] to synthesize the DAU for five processes and five resources with

the QualCore Logic .25μm standard cell library [14]. The Synthesis result is shown in Table 1. The "Total Area" column denotes the area in units equivalent to a minimum-sized two-input NAND gate in the library, and "# steps" means the worst case number of steps. In case where an SoC contains four PowerPC 755 PEs (1.7M gates each) and 16MB memory (33.5M gates), the area overhead in the SoC due to the DAU is about .005%.

Module Name	Lines of Verilog	Total Area	# Steps in Detection	# Steps in Resolution
DDU 5x5	203	364	6	–
Others in Figure 5	344	1472	–	8
Total	547	1836(.005%)	–	$6 \times 5 + 8 = 38$
SoC		40.344M	–	

Table 1: Synthesized result of the DAU.

6. EXPERIMENTS

6.1 Simulation Environment Setup

The experimental simulations were carried out using Seamless Co-Verification Environment (CVE) [13] aided by Synopsys VCS^{TM} for Verilog HDL simulation and XRAY^{TM} for software debugging. We used Atalanta RTOS version 0.3 [9], a shared-memory multiprocessor RTOS. The RTOS code resides in the shared memory, and all PEs execute the same RTOS code and share kernel structures as well as the states of all processes and resources.

6.2 Experimental System

For the experimental simulations, we implemented in Verilog HDL the MPSoC with four processing elements (PEs) and four resources introduced in Figure 4 (except PE cores, which are typically provided by simulation tool vendors such as processor support packages from Seamless CVE [13]). The MPSoC has four Motorola MPC755s as PEs. Each MPC755 has separate instruction and data L1 caches each of size 32KB. The MPSoC of Figure 4 also has 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 MPSoC has a DAU for five processes and five resources, an arbiter and 16MB of shared memory. The master clock period of the bus system is 10 ns. Code for each MPC755 runs on an instruction-accurate (not cycle-accurate) MPC755 simulator provided by Seamless CVE [13]. As mentioned in Example 3, we invoke one process on each PE and prioritize all processes, p_1 being the highest and p_4 being the lowest.

6.3 Application Example I

We show a sequence of requests and grants that would lead to grant deadlock (G-dl) as shown in Figure 6 and Table 2. Recall that there is no constraint on the ordering of the resource usage. That is, when a process requests a resource and the resource is available, it is granted immediately to the requesting process. At time t_1, process p_1, running on PE1, requests both VI and MPEG, which are then granted to p_1. After that, p_1 starts receiving a video stream through VI and does MPEG processing. At time t_2, process p_3, running on PE3, requests MPEG and WI to convert a frame to an image and to send the image through WI. However, only WI is granted to p_3 since MPEG is unavailable. At time t_3, p_2 running on PE2 also requests MPEG and WI, which are not available for p_2. When MPEG is released by p_1 at time t_4, MPEG would typically (assuming the DAU is not used) be granted to p_2 since p_2 has a priority higher than p_3; thus, the system would typically end up in deadlock. However, the DAU checks the potential G-dl and then avoids the G-dl by granting MPEG to p_3 even though p_3 has a priority lower than p_2. Then, p_3 uses and releases MPEG and WI at time t_6. After that, MPEG and WI are granted to p_2 at time t_7, which finishes its job at time t_8.

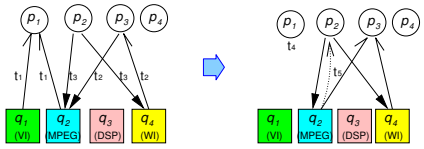

Figure 6: Events RAG (G-dl).

Time	Events
t_0	The application starts.
t_1	p_1 requests q_1 and q_2, which are granted to p_1 immediately.
t_2	p_3 requests q_2 and q_4; only q_4 is granted to p_3 since q_2 is not available.
t_3	p_2 also requests q_2 and q_4.
t_4	q_1 and q_2 are released by p_1.
t_5	Then, the DAU tries to grant q_2 to p_2 since p_2 has a priority higher than p_3. However, the DAU detects potential G-dl. Thus, the DAU grants q_2 to p_3, which does not lead to a deadlock.
t_6	q_2 and q_4 are used and released by p_3.
t_7	q_2 and q_4 are granted to p_2.
t_8	p_2 finishes its job, and the application ends.

Table 2: A sequence of requests and grants that could lead to grant deadlock (G-dl).

With the above scenario, we wanted to measure two figures, the average execution time of deadlock avoidance algorithms and the total execution time of the application in two cases: (i) using the DAU versus (ii) using DAA (Algorithm 3) in software.

6.4 Experimental Result I

Table 3 shows that the DAU achieves a 312X speed-up of the average algorithm execution time and gives a 37% speed-up of application execution time over avoiding deadlock with DAA in software. Note that during the run-time of the application, the deadlock avoidance algorithms were invoked 12 times, respectively (since every request and release invokes one of the algorithms).

Method of Implementation	Algorithm Run Time	Application Run Time	Speedup
DAU(hardware)	7	34791	$\frac{47704-34791}{34791} = 37\%$
DAA in software	2188	47704	

*The unit is a clock, and the values are averaged. The speedup is calculated according to the formula by Hennessy and Patterson [10].

Table 3: Execution time comparison (G-dl).

6.5 Application Example II

We show a sequence of requests and grants that would lead to request deadlock (R-dl) as shown in Figure 7. In this example, we assume the following: (i) Process p_1 requires resources q_1 (VI) and q_2 (MPEG) to complete its job. (ii) Process p_2 requires resources q_2 (MPEG) and q_3

(DSP). (iii) Process p_3 requires resources q_3 (DSP) and q_1 (VI). The detailed sequence is shown in Table 4. At time t_6, when process p_1 requests q_2, request deadlock (R-dl) would occur. However, the DAU detects the potential R-dl and then avoids the R-dl by asking p_2 to give up resource q_2 since p_1 has a priority higher than p_2, which is the current owner of q_2. As a result, at time t_7, p_2 gives up and releases q_2, which is going to be granted to p_1 (of course, p_2 has to request q_2 again). After using q_1 and q_2, p_1 releases q_1 and q_2 at time t_8. While q_1 is going to be granted to p_3, q_2 is going to be granted to p_2. Thus, p_3 uses q_1 and q_3 and then releases q_1 and q_3 at time t_9; q_3 is granted to p_2, which then uses q_2 and q_3 and finishes its job at time t_{10}.

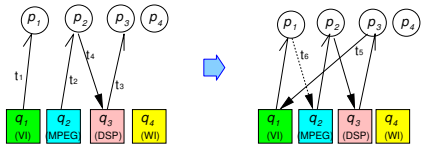

Figure 7: Events RAG (request deadlock).

Time	Events
t_0	The application starts.
t_1	p_1 requests q_1; q_1 is granted to p_1.
t_2	p_2 requests q_2; q_2 is granted to p_2.
t_3	p_3 requests q_3; q_3 is granted to p_3.
t_4	p_2 requests q_3, which becomes pending.
t_5	p_3 requests q_1, which also becomes pending.
t_6	p_1 requests q_2, which is about to lead to R-dl. However, the DAU detects the possibility of R-dl. Thus, the DAU asks p_2 to give up resource q_2.
t_7	p_2 releases q_2, which is granted to p_1. A moment later, p_2 requests q_2 again.
t_8	p_1 uses and releases q_1 and q_2. Then, while q_1 is granted to p_3, q_2 is granted to p_2.
t_9	p_3 uses and releases q_1 and q_3, which are granted to p_2.
t_{10}	p_2 finishes its job, and the application ends.

Table 4: A sequence of requests and grants that would lead to request deadlock (R-dl).

We similarly measured two figures, the average execution time of deadlock avoidance algorithms and the total execution time of the application in two cases: (i) exploiting the DAU and (ii) using DAA in software.

6.6 Experimental Result II

Table 5 demonstrates that the DAU achieves a 294X speed-up of the average algorithm execution time and gives a 44% speed-up of application execution time over avoiding deadlock with DAA in software. Note that during the run-time of the application, the deadlock avoidance algorithms were invoked 14 times, respectively.

Method of Implementation	Algorithm Run Time	Application Run Time	Speedup
DAU(hardware)	7.14	38508	$\frac{55627-38508}{38508} = 44\%$
DAA in software	2102	55627	

*The unit is a clock, and the values are averaged.

Table 5: Execution time comparison (R-dl).

7. CONCLUSION

A novel Deadlock Avoidance Algorithm (DAA) and its hardware implementation in the Deadlock Avoidance Unit (DAU) are described in this paper. The DAU provides a very fast and very low area way of avoiding deadlock at run-time, which helps free programmers from worrying about deadlock. Whenever a request occurs in a system, the DAU checks for the possibility of request deadlock (R-dl); if the request would lead to R-dl, then the DAU avoids the R-dl by asking a lower priority process that is holding resource(s) in an R-dl chain to give up resource(s). Whenever a resource is released and needs to be granted, the DAU quickly detects the possibility of grant deadlock (G-dl) and then resolves the situation by granting a released resource to an appropriate process such that the grant does not cause G-dl. We demonstrated the following with two examples: (i) The DAU automatically avoided deadlocks as well as reduced the deadlock avoidance time by 99% *(about 300X)* as compared to DAA in software. (ii) The DAU achieved a 37% speed-up of application execution time as compared to the execution time of the same application that uses DAA in software. While our examples are not industrial strength full product code, nevertheless we expect similar results as MPSoC designs become more commonplace; we predict that our DAU can potentially help especially in real-time scenarios. For our future work, we will extend this work to the design of various sizes of DAU and also support systems with non-priority based scheduling as well, e.g., round-robin.

Acknowledgment

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We would like to acknowledge donations received from Denali, HP, Intel, QualCore, Mentor Graphics, National Semiconductor, Sun and Synopsys.

8. REFERENCES

[1] Xilinx, http://www.xilinx.com/.
[2] E. Coffman, M. Elphick and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, pp. 67–78, June 1971.
[3] E. Dijkstra, "Cooperating sequential processes," Tech. Rep. EWD-123, Technological University, Eindhoven, The Netherlands, Sep. 1965.
[4] 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.
[5] J. Lee and V. Mooney, "An O(min(m,n)) parallel deadlock detection algorithm," Tech. Rep. GIT-CC-03-41, College of Computing, Georgia Tech, Atlanta, GA, Sep. 2003, http://www.coc.gatech.edu/research/pubs.html.
[6] F. Belik, "An efficient deadlock avoidance technique," *IEEE Trans. on Computers*, 39(7), pp. 882–888, July 1990.
[7] N. Gebraeel and M. Lawley, "Deadlock detection, prevention and avoidance for automated tool sharing systems," *IEEE Trans. on Robotics and Automation*, 17(3) pp. 342–356, June 2001.
[8] J. Ezpeleta, F. Tricas, Garcia-Valles and J. Colom, "A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states," *IEEE Trans. on Robotics and Automation*, 18(4) pp. 621–625, Aug. 2002.
[9] 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, College of Computing, Georgia Tech, Atlanta, GA, 2002, http://www.coc.gatech.edu/research/pubs.html.
[10] J. Hennessy and D. Patterson, *Computer architecture - a quantitative approach*. Morgan Kaufmann Publisher, Inc., San Francisco, CA, 1996.
[11] R. Holt, "Some deadlock properties of computer systems," *ACM Computing surveys*, pp. 179–196, Sep. 1972.
[12] Design Compiler, http://www.synopsys.com/products/logic/logic.html.
[13] Mentor Graphics, Hardware/Software Co-Verification: Seamless, http://www.mentor.com/seamless/.
[14] QualCore Logic. http://www.qualcorelogic.com/.