A Novel Parallel Deadlock Detection Algorithm and Architecture

Pun H. Shiu, YuDong Tan, Vincent J. Mooney III
Electrical and Computer Engineering, Georgia Institute of Technology
{ship, ydtan, mooney}@ece.gatech.edu

ABSTRACT

A novel deadlock detection algorithm and its hardware implementation are presented in this paper. The hardware deadlock detection algorithm has a run time complexity of $O_{hw}(\min(m,n))$, where m and n are the number of processors and resources, respectively. Previous algorithms based on a Resource Allocation Graph have $O_{sw}(m \times n)$ run time complexity for the worst case. We simulate a realistic example in which the hardware deadlock detection unit is applied, and demonstrate that the hardware implementation of the novel deadlock detection algorithm reduces deadlock detection time by 99.5%. Furthermore, in a realistic example, total execution time is reduced by 68.9%.

Keywords

Deadlock Detection, Parallel Algorithm, Hardware/Software Codesign, Real-Time Operating System.

1. INTRODUCTION

Development of a real-time System-on-a-Chip (SoC) demands a deterministic and fast Real-Time Operating System (RTOS), which provides service(s) and manages resource(s) between software and hardware. However, the algorithms implementing RTOS services may be non-deterministic or may have long execution times. Since the RTOS also competes for the shared CPU, RTOS services may be even less deterministic. For real-time systems, optimization beyond assembly code is desired, such as a custom hardware unit similar to FASTCHART[6]. Therefore, implementing deadlock detection in hardware may provide a better alternative, which not only reduces the load of a shared CPU but also improves determinism of the overall SoC system.

An RTOS includes a job scheduler, interprocess communication, and resource allocation, thereby providing a means for applications to have several concurrent tasks. Moving deadlock detection out of the RTOS and into custom hardware gives more bandwidth to the rest of the RTOS services, allowing the RTOS to handle more services with faster run time, more concurrency, and better utilization of the underlying SoC.

Furthermore, future SoC designs are likely to include 4-10 heterogeneous processors together with 10-50 on-chip hardware resources such as an ethernet interface, Bluetooth and other wireless devices, Viterbi filtering, FFT, Direct Memory Access (DMA), and many more. Ideally, programmers of these SoC devices would only write deadlock-free code. However, if they do not, this paper presents a very fast and very low area (206 nanoseconds and 14,142 gates for 50 requestors and 50 resources) hardware deadlock detection unit which programmers can use to check for deadlock at run-time. If deadlock is detected, the programmer can include code for resolving the situation, for example by releasing held resources.

The proposed hardware deadlock detection implementation allows deadlock detection to be run in a deterministic manner (short latency), so that the system can meet its deadlines. The contribution of this work is an enhanced and deterministic capability of dynamic (run-time) detection of deadlocks. The run time complexity of the proposed hardware algorithm is $O_{hw}(\min(m,n))$, where m is the number of processors and n is the number of resources. Previous algorithms have run time complexities of $O_{sw}(m \times n)$[1], $O_{sw}(m \times n)$[4], and $O_{sw}(e)$[3], where e is the number of edges (the sum of the number of requests and grants) for a time-shared operating system.

Section 2 reviews some previous work on deadlock detection. Section 3 presents the new deadlock detection algorithm. Section 4 illustrates how parallelism is implemented in the hardware architecture of the proposed algorithm. Section 5 presents a practical application example. Section 6 concludes this paper.

2. BACKGROUND

Informally speaking, a deadlock is a system state where requestors are waiting for resources held by other requestors which, in turn, are also waiting for some resources held by the previous requestors. In this paper, we only consider the case where requestors are processors on a single piece of silicon (SoC). A deadlock situation results in permanently blocking a set of processors from doing any useful work.

There are four necessary conditions which allow a system to deadlock[9]: (a) Non-Preemptive: resources can only be released by the holding processor; (b) Mutual Exclusion: resources can only be accessed by one processor at a time; (c) Blocked Waiting: a processor is blocked until the resource becomes available; and (d) Hold-and-Wait: a processor is

.

using resources and making new requests for other resources at the same time, without releasing held resources until some time after the new requests are granted.

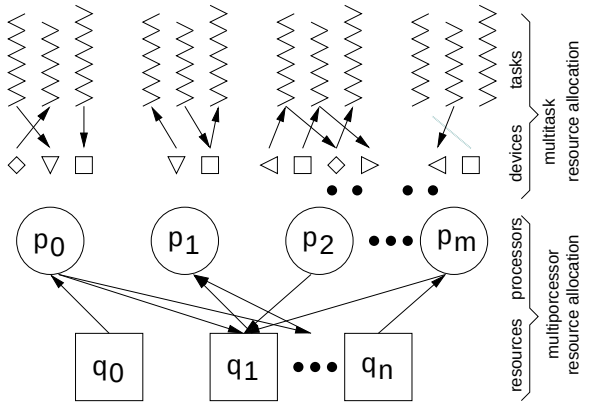

Figure 1: Bipartite Graphic Representation.

Deadlock detection can be represented by a Resource Allocation Graph (RAG), commonly used in operating systems. A RAG is defined as a graph (V, E) where V is a set of nodes and E is a set of ordered pairs or edges (v_i, v_j) such that $v_i, v_j \in V$. V is further divided into two disjoint subsets: $P = \{p_0, p_1, p_2, \cdots, p_m\}$ where P is a set of processor nodes shown as circles in Figure 1; and $Q = \{q_0, q_1, q_2, \cdots, q_n\}$ where Q is a set of resource nodes shown as boxes in Figure 1. A RAG is a graph bipartite in the P and Q sets. An edge $e_{ij} = (p_i, q_j)$ is a request edge if and only if $p_i \in P, q_j \in Q$. An edge $e_{ji} = (q_j, p_i)$ is a grant edge if and only if $p_i \in P, q_j \in Q$. The maximum number of edges in a RAG is $m \times n$. A node is a sink when a resource (processor) has only incoming edge(s) from processor(s) (resource(s)). A node is a source when a resource (processor) has only outgoing edge(s) to processor(s) (resource(s)). A path is a sequence of edges $\varepsilon = \{(p_{i_1}, q_{j_1}), (q_{j_1}, p_{i_2}), ..., (p_{i_k}, q_{j_k}), (q_{j_k}, p_{i_{k+1}}), ..., (q_{j_s}, p_{i_{s+1}})\}$ where $\varepsilon \subseteq E$. If a path starts from and ends at the same node, then it is a cycle. A cycle does not contain any sink or source nodes.

In Figure 1, each processor has more than one task running on the processor. Each resource can be accessed by only one processor at a time. A downward arrow indicates a processor making a request to a resource, while an upward arrow indicates that the processor is holding and using the granted resource. For example, $q_0 \rightharpoonup p_0$ represents that processor p_0 is granted (upward arrow) resource q_0.

The focus of this paper is deadlock detection. For our hardware deadlock detection implementation, we make three assumptions. First, each resource type has one unit. Thus, a cycle is a sufficient condition for a deadlock[2]. Second, a satisfiable request will be granted immediately, making the overall system expedient[2]. Thus, a processor is blocked only if it cannot obtain the requested resource. Third, each processor can make multiple requests at the same time.

Software deadlock detection algorithms have been proposed in the past. All proposed algorithms, including those based on a RAG, have $O_{sw}(m \times n)$ for the worst case. An $O_{sw}(m \times n^2)$ run time complexity detection algorithm is proposed by Shoshani[1]. Holt proposed an $O_{sw}(m \times n)$ algorithm to detect knots[2]. Both Holt and Shoshani's algorithms are based on a RAG representation. Leibfried proposed a system state using an adjacency matrix representation and a corresponding scheme which detects deadlock with matrix multiplications, but the run time complexity is $O_{sw}(m^3)$. Kim[4] proposed an algorithm with $O_{sw}(1)$ for detection and $O_{sw}(m \times n)$ for detection preparation. In this paper, we propose a hardware algorithm with $O_{hw}(\min(m,n))$ based on a new matrix representation. The order notation used in software and hardware is distinguished by $O_{sw}(\cdots)$ and $O_{hw}(\cdots)$. The proposed hardware deadlock detection algorithm makes use of parallelism and can handle multiple requests/grants, making the proposed algorithm faster than the $O_{sw}(1)$ algorithm[4], which actually has $O_{sw}(m \times n)$ latency in multiprocessor systems. Therefore, the SoC parallel deadlock detection algorithm/hardware is faster than any previously reported algorithms.

3. A NEW ALGORITHM FOR DEADLOCK DETECTION

In this section, we will first introduce the matrix representation of a deadlock detection problem. The new algorithm is based on this matrix representation. Next, we present some essential features of the proposed algorithm. This algorithm is parallel, and thus can be mapped into a hardware architecture which can handle multiple requests/grants simultaneously and can detect multiple deadlocks in linear time, hence, significantly improving performance.

3.1 Matrix Representation of A Deadlock Detection Problem

In graph theory, any directed graph can be represented with an adjacency matrix[8]. Thus, we can represent a RAG with an adjacency matrix. However, there are two kinds of edges in a RAG: grant edges, which point from resources to processors; and request edges, which point from processors to resources. To distinguish different edges, we designate elements in the adjacency matrix with three different values as shown in Figure 2. This Figure shows the matrix representation of a given system with processors p_1, p_2, ..., p_i, ..., p_m and resources q_1, q_2, ..., q_j, ..., q_n. The leftmost column is the processors' label column. The top row is the resources' label row. If there is a request edge (p_i, q_j) in the RAG, the corresponding element in the matrix is r. If there is a grant edge (q_i, p_j) in the RAG, the corresponding element in the matrix is g. Otherwise, the value of the element is 0.

P\Q	q_1	q_2	$\cdots$	q_n
p_1	g	r	$\cdots$	0
p_2	r	g	$\cdots$	0
p_3	0	r	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
p_m	0	r	$\cdots$	g

Figure 2: Matrix Representation.

This variant of the adjacency matrix of a RAG (V, E) can be defined formally as follows:

$M = [m_{ij}]^{m \times n}$, $(1 \leq i \leq m, 1 \leq j \leq n)$, where m is the number of processors and n is the number of resources.

$m_{ij} \in \{r, g, 0\}$

$m_{ij} = r$, *if* $\exists(p_i, q_j) \in E$
$m_{ij} = g$, *if* $\exists(q_j, p_i) \in E$
$m_{ij} = 0$, *otherwise*

This matrix provides a template able to represent all request and grant combinations. Note that each resource has at most one grant, that is, there is at most one g in a column at any time. However, there is no constraint on the number of requests from each processor.

If there are deadlocks in a system, there must be at least one cycle in its RAG, that is, there must be a sequence of edges, $\varepsilon = \{(p_{i_1}, q_{j_1}), (q_{j_1}, p_{i_2}), ..., (p_{i_k}, q_{j_k}), (q_{j_k}, p_{i_{k+1}}), ..., (p_{i_s}, q_{j_s}), (q_{j_s}, p_{i_1})\}$, where $\varepsilon \subseteq E$ (see[10] for a detailed proof). In the matrix representation, this cycle is mapped into a sequence of matrix elements $\omega = \{m_{i_1 j_1}, m_{i_2 j_1}, ..., m_{i_k j_k}, m_{i_{k+1} j_k}, m_{i_{k+1} j_{k+1}}, ..., m_{i_s j_s}, m_{i_1 j_s}\}$ where $m_{i_1 j_1}$, $m_{i_2 j_2}$, ..., $m_{i_k j_k}$, ..., $m_{i_s j_s}$ are requests (r's) and $m_{i_2 j_1}, m_{i_3 j_2}$, ..., $m_{i_{k+1} j_k}$, ..., $m_{i_1 j_s}$ are grants (g's). By this fact, we can detect deadlocks in a system with its adjacency matrix. Next, we will present the new detection algorithm.

EXAMPLE 1. Matrix representation of a given system
Let us consider the system presented in Figure 1. Its adjacency matrix is presented in Figure 2. There is a cycle in the RAG of this system, that is, $\{(p_1, q_2), (q_2, p_2), (p_2, q_1), (q_1, p_1)\}$. In the adjacency matrix of this system, this cycle is indicated by a sequence of non-zero elements which is $\{m_{12}, m_{22}, m_{21}, m_{11}\}$, where $m_{12} = m_{21} = r$, $m_{11} = m_{22} = g$. So there is a deadlock in this case. □

3.2 A New Hardware Deadlock Detection Algorithm

On the basis of the matrix representation, we propose a novel hardware deadlock detection algorithm. The basic idea in this algorithm is iteratively reducing the matrix by removing those columns or rows corresponding to any of the following cases:

(i) a row or column of all 0's;

(ii) a source (a row with one or more r's but no g's, or a column with one g and no r's);

(iii) a sink (a row with one or more g's but no r's, or a column with one or more r's but no g's).

This continues until the matrix cannot be reduced any more. At this time, if the matrix still contains row(s) or column(s) in which there are non-zero elements, then there is at least one deadlock. Otherwise, there is no deadlock. (See [10] for a detailed proof.) The description of this algorithm is shown in Algorithm 3.1 .

ALGORITHM 3.1. *Parallel Deadlock Detection*
Step 0: Initialization
$M = [m_{ij}]^{m \times n}$,
where $m_{ij} \in \{r, g, 0\}, (i = 1, \ldots, m$ *and* $j = 1, \ldots, n)$
$m_{ij} = r$, *if* $\exists(p_i, q_j) \in E$.
$m_{ij} = g$, *if* $\exists(q_j, p_i) \in E$.
$m_{ij} = 0$, *otherwise.*
$\Lambda = \{m_{ij} \mid m_{ij} \in M, m_{ij} \neq 0\}$;

Note that Λ *is a set consisting initially of all the non-zero entries in the matrix* M.

Step 1: Remove all sinks and sources
DO {
reducible $= 0$;
For each column:
if $(\exists m_{ij} \in \Lambda \mid \forall k, k \neq i, m_{kj} \in \{m_{ij}, 0\})\{$
$\Lambda_{column} = \Lambda - \{m_{ij} \mid j = 1, 2, 3, \ldots, m\}$,
reducible $= 1$;
} *else* { }
For each row:
if $(\exists m_{ij} \in \Lambda \mid \forall k, k \neq j, m_{ik} \in \{m_{ij}, 0\})\{$
$\Lambda_{row} = \Lambda - \{m_{ij} \mid i = 1, 2, 3, \ldots, n\}$,
reducible $= 1$;
} *else* { }
$\Lambda = \Lambda_{column} \cap \Lambda_{row}$;
} *UNTIL (reducible* $= 0$*);*

Step 2: Detect Deadlock
if $(\Lambda \neq \emptyset)$, *then deadlock exists.*
if $(\Lambda = \emptyset)$, *then no deadlock exits.*

The following example illustrates how the algorithm works. In each iteration of this parallel algorithm, at least one reduction can be performed if the matrix is reducible. Hence, it takes at most min(m,n) iterations to complete the deadlock detection.

EXAMPLE 2. Two processors and three resources

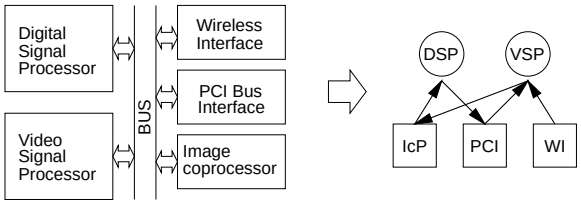

Figure 3: SoC Example

This example has two processors: DSP and Video Signal Processor, as p_1 and p_2 respectively. The devices are Image Co-processor, PCI, and Wireless Interface, as q_1, q_2, and q_3 respectively as shown in Figure 3.

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

Table 1: Small Example with 2 Processes and 3 Resources.

The matrix representation of this example is shown in Table 1. In this matrix, the first and second column contain both g and r, and hence are not reducible. However, the third column contains only g. Thus $m_{12} = g$ can be reduced. At the same time, each row is also examined, however there is no reduction possible. Since there is one reduction, the next iteration will be carried out. In the second iteration, the first and second columns still contain both g and r, and hence are not reducible. At the same time, each row is also checked, but no reduction is possible for any row. Since there are no more reductions, a conclusion is drawn. In this case, hardware deadlock detection takes two iterations and finds a deadlock.

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

Table 2: Small Example without Deadlock

Let us remove the edge (p_2, q_2) in this case and consider it again. The matrix is shown in Table 2. In this matrix, the first column cannot be reduced, because of the existence of both g and r, while the second and third columns can be reduced, because the second column has only one r and no g's, and the third column has only one g and no r's. At the same time, the first and second rows cannot be reduced, because of the existence of both g and r in each row. Since this iteration has a reduction, Step 1 will be re-executed with the

second and third columns having been removed. During the second iteration, the first column is not reduced, because there are both r and g in this column. However, the first row can be reduced because only g is in this row, and the second row can be also reduced because only r is in this row. Then Step 1 is executed again in what is now a third iteration of the Hardware Deadlock Detection Algorithm. There are no more reductions, because the matrix now is empty. Step 2 concludes that there is no deadlock. In this case, three iterations are taken to complete detection. □

4. ARCHITECTURE

We will illustrate how the parallelism of the new algorithm works cycle by cycle. Our architecture will be able to perform in parallel all calculations from any particular cycle.

EXAMPLE 3. Detail Calculation of Example 2

Consider Table 1 from Example 2. For this case, we have the following after Step 0 (initialization) of the new algorithm:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{bmatrix} = \begin{bmatrix} g & r & 0 \\ r & g & g \end{bmatrix}$$

$$\Lambda = \{m_{11},\ m_{12},\ m_{21},\ m_{22},\ m_{23}\}$$

Now we will show all the calculations that occur in the first clock cycle of the architecture for this algorithm. To explain this, first we replace r with 10, g with 01, and 0 with 00. The result is as follows:

$$M_c = \begin{bmatrix} 01 & 10 & 00 \\ 10 & 01 & 01 \end{bmatrix} \quad M_r = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

M_r is a different representation of M where we have represented r as $\begin{smallmatrix}1\\0\end{smallmatrix}$, g as $\begin{smallmatrix}0\\1\end{smallmatrix}$, and 0 as $\begin{smallmatrix}0\\0\end{smallmatrix}$. Now we perform in parallel, a bitwise OR on each individual column in M_c (note that now there are six columns) and each individual row in M_r. The result is as follows:

$$M_c = \begin{bmatrix} 01 & 10 & 00 \\ 10 & 01 & 01 \end{bmatrix} \quad M_r = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = M_{RBO}$$

$$M_{CBO} = \begin{bmatrix} 11 & 11 & 01 \end{bmatrix}$$

The vector M_{CBO} is the result of the Column Bitwise OR. The vector M_{RBO} is the result of Row Bitwise OR. Next we perform exclusive-OR on every two bits in the M_{CBO} and M_{RBO}, with the following result:

$$XOR_{\text{below}} = [\ 1 \oplus 1,\quad 1 \oplus 1,\quad 0 \oplus 1\ ] = [\ 0\quad 0\quad 1\ ],$$

$$XOR_{\text{right}} = [\ 1 \oplus 1,\quad 1 \oplus 1\ ]^T = [\ 0\quad 0\ ]^T.$$

The XOR_{below} is the XOR on every two bits of M_{CBO} and the XOR_{right} is the XOR of every two bits of M_{RBO}. Finally, all m_{ij} corresponding to any 1 in either XOR result are removed from Λ. In this case, m_{23} is removed, resulting in $\Lambda = \{m_{11}, m_{12}, m_{21}, m_{22}\}$. □

Continuing in this way, Λ is reduced each clock cycle. Whenever no more reductions are possible (which occurs in at most $\min(m, n)$ steps), we are done. If Λ is empty, there is no deadlock; otherwise, deadlock exists.

The architecture for a system with 3 processors and 3 resources is shown in Figure 4. The architecture is able to perform all of the calculations for each step of the algorithm in minimal time. The architecture is explained in great detail in [10].

4.1 Synthesized Result of Deadlock Detection Unit (DDU)

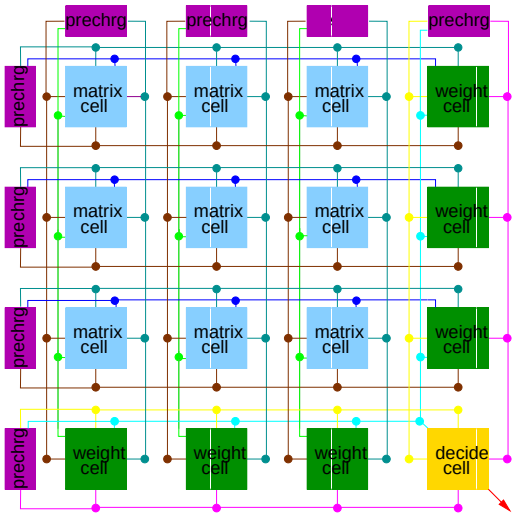

Figure 4: Architecture of DDU

We used the Synopsys Design Compiler (DC) to synthesize DDU with the AMI[13] 0.3μm standard cell library. The results are shown in Table 3. The "Area" column denotes the area in units equivalent to a minimum-sized two-input NAND gate in the AMI 0.3μm standard cell library. Note that the hardware area cost is $O(m \times n)$. For an example of 50 processors and 50 resources, the worst delay for deadlock detection is 206ns, assuming each step is calculated in 4.12ns (which requires a 242MHz clock).

# Processor × # Resources	Lines of Verilog	Area	Delay Per One Step(ns)	Worst Case # Steps	Worst Case Delay(ns)
2x3	49	186	0.91	2	1.82
5x5	73	364	2.21	5	11.05
7x7	102	455	2.51	7	17.57
10x10	162	622	3.66	10	36.6
50x50	2682	14142	4.12	50	206

Table 3: Synthesis Results of DDU

5. EXPERIMENTAL RESULTS

We simulated a large number of deadlock detection cases with hardware and software respectively. Figure 5 shows our measurements for the four processor and four resource case simulated in Seamless CVE with up to nineteen edges in the RAG. In the legend of Figure 5, "Dead SW" represents deadlocked scenarios detected by the software algorithm. "Live SW" denotes non-deadlocked scenarios detected by the software algorithm. "Dead HW" and "Live HW" mean, respectively, deadlocked and non-deadlocked scenarios detected by the hardware algorithm. Notice that in all cases, three orders of magnitude separate software and hardware execution times.

The software deadlock detection algorithm spends a lot of time searching linked lists, manipulating matrices, and updating data structures. In real time systems, this three

orders of magnitude difference in detection time can allow the system to fix itself, where, without the DDU, the system could miss its real-time deadline by the time it has discovered deadlock using the software algorithm. Figure 5 empirically verifies that the hardware deadlock detection algorithm is $O(min(m, n))$ as was explained earlier.

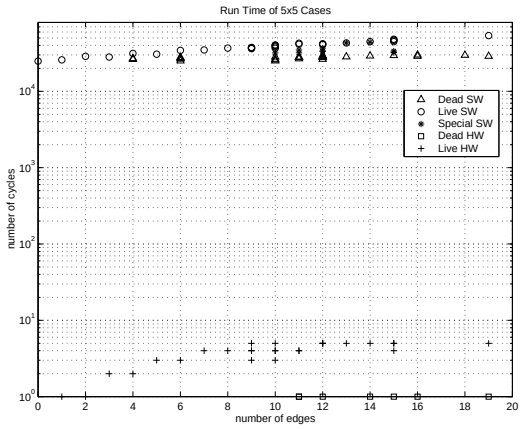

Figure 5: Comparison of Run Time Performance of Deadlock Detection Using PowerPC and Hardware DDU

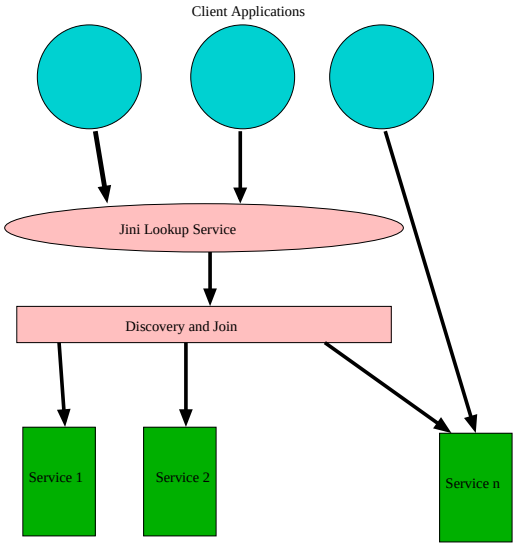

Figure 6: Jini Lookup Service System

However, the above simulation only measure the detection time. To verify the functionality of the Deadlock Detection Unit (DDU for short) and analyze its performance in applications, we also applied the DDU in a real-life example which is shown in Figure 6. This is a Jini lookup service system [14], in which client applications can request services through intermediate layers (i.e., lookup, discovery and join).

We simulate this system with four processors and four hardware units. Application tasks which run on processors request the service of hardware units (PCI, MPEG, FFT and Wireless Interface(WI)). Functions of intermediate layers are partially implemented by an RTOS (we choose the δ RTOS developed at Georgia Tech in this simulation). Figure 7 shows the architecture of the system simulation.

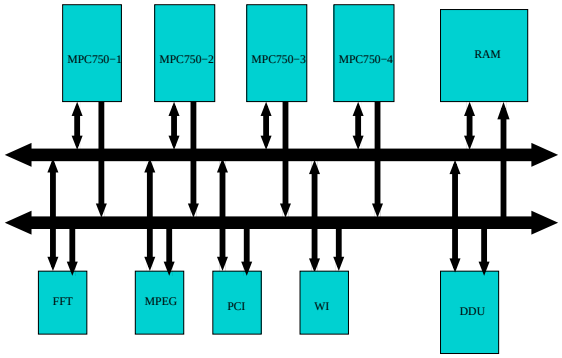

Figure 7: An SoC Architecture with Four Processors and Four Resources

The first processor processes video streams. The second processor completes some signal processing algorithms. The third processor handles services such as fax, voice and email. The fourth processor handles communication functions. Some functionalities in these tasks are implemented with hardware, so they will request the services of the hardware units. Because this is a system with multiple processors and multiple resources, it is possible that a deadlock may occur. Now, we give an example sequence of requests and grants which leads to a deadlock. Table 4 is the description of events in this sequence.

Time	Events No.	Events
t_1	e_1	MPC750-1 requests FFT, MPEG; FFT and MPEG are granted to MPC750-1 immediately
t_2	e_2	MPC750-3 requests FFT, PCI; PCI is granted to MPC750-3 immediately
t_3	e_3	MPC750-2 requests FFT, MPEG
t_4	e_4	FFT is released by MPC750-1
t_5	e_5	FFT is granted to MPC750-2

Table 4: An Example Sequence

We can represent the state of the system at every time instant in Table 4 with adjacency matrices, which are shown in Table 5.

At time instant t_5, there is a deadlock in the system. Figure 8 shows the events sequence in this example. Here, we measure execution time in CPU clock cycles, which in our case is a Motorola PowerPC 750 (MPC750) with an 83.3 MHz clock, with the DDU on the same clock (83.3MHz). This example was simulated in Seamless CVE.

The times shown are not from an actual industrial product but instead are estimates intended to exemplify an application with short execution times. Suppose we start deadlock detection after event e_5 occurs and the deadlock detection time is Δ, what we are interested in is the values of Δ with different deadlock detection methods. We compare the impact on deadlock detection time and total execution time of both methods: software deadlock detection versus hardware deadlock detection.

Table 6 shows the results.

Q\P (t_1)	MPC750			
	p_1	p_2	p_3	p_4
FFT, q_1	g	0	0	0
MPEG, q_2	g	0	0	0
PCI, q_3	0	0	0	0
WI, q_4	0	0	0	0

Q\P (t_2)	MPC750			
	p_1	p_2	p_3	p_4
q_1	g	0	r	0
q_2	g	0	0	0
q_3	0	0	g	0
q_4	0	0	0	0

Q\P (t_3)	MPC750			
	p_1	p_2	p_3	p_4
q_1	g	r	r	0
q_2	g	0	0	0
q_3	0	r	g	0
q_4	0	0	0	0

Q\P (t_4)	MPC750			
	p_1	p_2	p_3	p_4
q_1	0	r	r	0
q_2	g	0	0	0
q_3	0	r	g	0
q_4	0	0	0	0

Q\P (t_5)	MPC750			
	p_1	p_2	p_3	p_4
q_1	0	g	r	0
q_2	g	0	0	0
q_3	0	r	g	0
q_4	0	0	0	0

Table 5: Adjancey Matrices of the Example

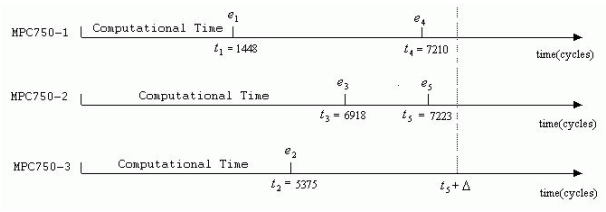

Figure 8: Events Sequence in the Example

The total execution time is reduced by 16,036 clock cycles, an improvement of 68.9%. This reduction in deadlock detection time may allow a real-time deadline to be met which would be missed with the software deadlock detection algorithm.

6. CONCLUSION

A deadlock detection algorithm is implemented in hardware using a newly reported parallel algorithm. The deadlock detection hardware algorithm has $O_{hw}(\min(m,n))$ run time complexity, an improvement of approximately three orders of magnitude in practical cases. For multiprocessor programmers who do not write code formally proven to be deadlock free, the hardware deadlock detection unit provides a very fast and very low area way of checking for deadlock at run-time. In this way, programmers can quickly detect deadlock and then resolve the situation, e.g., by releasing held resources.

Method of Deadlock Detection	Detection Time Δ (Cycles)	$t_5+\Delta$
Software Algorithm	16038	23261
DDU	2	7225

Table 6: Deadlock Detection Time and Total Execution Time

7. ACKNOWLEDGMENTS

This research is funded by the Design Automation Conference Scholarship and by NSF under INT-9973120, CCR-9984808, and CCR-0082164.

8. REFERENCES

[1] A. Shoshani, E. G. Coffman, Jr., "Detection, Prevention and Recovery from deadlocks in multiprocess, multiple resource systems," Princeton University, Technical Report Number 80, October, 1969.

[2] R. C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, Vol. 4, No.3, September 1972.

[3] I. Cahit, "Deadlock detection using (0, 1)-labeling of resource allocation graphs," *IEE Proceedings: Computers and Digital Techniques*, Vol. 145, No. 1, pp. 68-72, January 1998.

[4] Ju Gyun Kim, "Algorithmic approach on deadlock detection for enhanced parallelism in multiprocessing systems," *Aizu International Symposium on Parallel Algorithms Architecture Synthesis*, IEEE, Piscataway, NJ (USA), 1997, pp. 233-238.

[5] Lennard Lindh, "FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel," *Euromicro workshop on Real-Time Systems*, June 1991.

[6] Lennard Lindh, "FASTCHART - Idea and Implementation," *International Conference on Computer Design (ICCD)*, Boston, USA, October 1991.

[7] Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd Ed, McGraw-Hill, 1978.

[8] Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[9] M. Maekawa, A. E. Oldehoeft and R. R. Oldehoeft, *Operating Systems*, Benjamin-Cummings Pub., 1987.

[10] P. H. Shiu and V. J. Mooney III, "The Principle of Parallel Deadlock Detection," Technical Report GIT-CC-00-30, http://www.cc.gatech.edu/tech_reports, December 2000.

[11] "Getting Started With Seamless Co-Verification Environment (Software Version 3.0-1.0)," Seamless Documentation, Mentor Graphics.

[12] "Virsim for Synopsys Reference Manual," Version 3.03 VCS 5.1 Beta 3, October 1999.

[13] American Microsystems Inc., http://www.amis.com.

[14] Steve Morgan, "Jini to the rescue," *IEEE Spectrum*, April 2000.