A Novel O(n) Parallel Banker's Algorithm for System-on-a-Chip

Jaehwan John Lee and Vincent John Mooney III
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
{jaehwan, mooney}@ece.gatech.edu

Abstract

This paper proposes a novel O(n) Parallel Banker's Algorithm (PBA) with a best-case run-time of O(1), reduced from an $O(mn^2)$ run-time complexity of the original Banker's Algorithm. We implemented the approach in hardware, which we call PBA Unit (PBAU), using Verilog HDL and verified the run-time complexity. PBAU is an Intellectual Property (IP) block that provides a mechanism of very fast, automatic deadlock avoidance for a MultiProcessor System-on-a-Chip (MPSoC, which we predict will be the mainstream of future high performance computing environments). Moreover, our PBA supports multiple-instance multiple resource systems. We demonstrate that PBAU not only avoids deadlock in a few clock cycles (1600X faster than the Banker's Algorithm in software) but also achieves in a particular example a 19% speedup of application execution time over avoiding deadlock in software. Lastly, the MPSoC area overhead due to PBAU is small, under 0.05% in our candidate MPSoC example.

1 Introduction

Recent trends show that System-on-a-Chip (SoC) technology enables multicore multithreaded systems on a single chip. An example of this is the Xilinx Vertex II Pro [1], which may contain multiple PowerPC processors and additional Intellectual Property (IP) cores. Furthermore, due to the ever increasing expansion of the Internet, a tremendous amount of multimedia related data is being created, edited and exchanged; this multimedia data is becoming larger with more varied and complicated encodings, requiring unprecedented processing power. To support such multimedia communication, numerous algorithms, specialized processors, image/video coding hardware modules and error isolation modules have been implemented and exploited [2].

Thus, we predict that in the near future, MultiProcessor SoC (MPSoC) designs will, as shown in Figure 1, have many Processing Elements (PEs) and hardware resources. In such future real-time MPSoCs, many processes will concurrently run and dynamically require and access such available on-chip resources. Not only that, but ensuring predictability and reliability in such MPSoCs will be much more difficult. In such systems, we predict that deadlock possibilities will no longer be ignorable issues, but will, if not properly addressed, become problems in the sharing of resources.

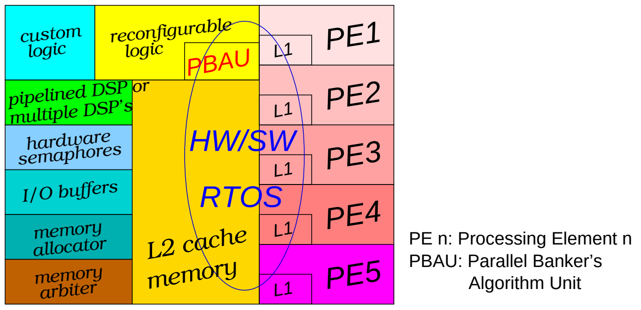

Figure 1 A practical MPSoC

Therefore, we propose a novel Parallel Banker's Algorithm and its hardware implementation and demonstrate its performance evaluation so that MPSoC programmers, who are reluctant to exploit deadlock avoidance approaches even as such approaches increase in importance, may be willing to adopt a faster hardware version of a deadlock avoidance approach.

2 Definitions and a Theorem

Definition 1 *A safe sequence is an enumeration* $p_1, p_2, \ldots, p_n$ *of all the processes in the system, such that for each* $i = 1, 2, \ldots, n$*, the resources that* p_i *may request are a subset of the union of resources that are currently available and resources currently held by* $p_1, p_2, \ldots, p_{i-1}$ *[3, 4].*

Theorem 1 *A system of processes and resources is in a safe state if and only if there exists a safe sequence* $\{p_1, p_2, \ldots, p_n\}$*. If there is no safe sequence, the system is in an unsafe state [4].*

If a system is in a safe state, completion of all the processes can be guaranteed by restricting resource usage in the system with a strategy – such as the Banker's Algorithm [3, 4] – which executes one of the safe sequences. An "unsafe" state is not necessarily a deadlocked state because there still may exist a possibility that all processes terminate successfully.

Definition 2 *A single-instance resource is a resource that services no more than one process at a time. That is, while the* *resource is processing a request from a process, all other processes requesting to use the resource must wait [5].*

Definition 3 *A multiple-instance resource is a resource that can service two or more processes at the same time, providing the same or similar functionality to all serviced processes [5].*

Example 1 An example of a multiple-instance resource
The SoC Dynamic Memory Management Unit (SoCDMMU) dynamically allocates and deallocates segment(s) of global level two (L2) memory between PEs with very fast and deterministic time (i.e., four clock cycles) [6]. In a system having an SoCDMMU and 16 segments of global L2 memory, which can be considered as a 16 instance resource, rather than having each PE (or process) keep track of each segment, PEs request segment(s) from the SoCDMMU (which keeps track of the L2 memory). In this way, not only can the overhead of tracking segments for each PE be reduced but also interfaces between PEs and segments can be simplified because PEs request segment(s) from one place (i.e., the SoCDMMU). ■

3 Previous Work and Motivation

In this section, we first mention related previous work and then introduce our approach.

The fundamental deadlock avoidance approach is the well-known Banker's Algorithm (BA) in the operating system realm. Dijkstra first introduced BA for single multiple-instance resource systems [3], and later Habermann improved it for multiple-instance multiple-resource systems [4]. In BA, each process declares the maximum possible number of instances for each resource it may need. Given this information, as each resource request is made, an assignment is authorized provided that there exists at least one sequence of executions that does not evolve to a deadlock. The run-time complexity of the Habermann's BA in software is $O(m \times n^2)$, where m and n are the numbers of resources and processes, respectively. The efficiency of the algorithm was later improved to $O(m \times n)$ by Holt [7]. Even though BA was proposed a few decades ago, minor variations to BA are still being proposed for critical systems that can greatly benefit from the algorithm. For instance, in 2002, J. Ezpeleta at al. proposed a banker's solution for deadlock avoidance in flexible manufacturing systems [8].

Recently, [9] has proposed a novel method of deadlock avoidance and its hardware implementation, which has a run-time complexity of $O(n \times min(m, n))$, where m and n are the numbers of resources and processes, respectively (see [9] for details). However, because the implementation of [9] is based on resource allocation graph [5] approach for single-instance resources, it can only be used for systems exclusively with single-instance resources. Our implementation, the Parallel Banker's Algorithm Unit (PBAU), on the contrary, can be used for not only a system with single-instance resources but also a system with multiple-instance resources as well.

4 Target System Model

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

Example 2 A future MPSoC
We refer to the device shown in Figure 1 as a particular MPSoC example. This MPSoC consists of five Processing Elements (PEs) and three resources – a counting semaphore with a group of I/O buffers, another counting semaphore with a group of multiple DSP processors and an SoCDMMU memory allocator [6] with a large L2 memory. Counting semaphores [3] are used to manage limited resources (including managing access to the resources). The MPSoC also contains a memory arbiter and a PBAU. PBAU in Figure 1 receives all requests and releases, decides whether or not the request can cause a deadlock and then permits the request only if no deadlock results. ■

We consider this kind of request-grant system with many resources and PEs shown in Figure 1 as our system model.

5 Methodology

Algorithm 1 shows our novel Parallel Banker's Algorithm (PBA) for multiple-instance multiple-resource systems. PBA executes whenever a process is requesting resources and returns the status of whether the request is successfully granted or rejected due to the possibility of deadlock. PBA decides if the system is still going to be sufficiently safe after the grant. Before explaining the details of PBA, let us first show data structures as shown in Table 1 and notations for PBA as shown in Table 2.

name	notation	explanation
Request[i][j]	R_{ij}	request from process i for resource j
Maximum[i][j]	X_{ij}	maximum demand of process i for resource j
Available[j]	V_j	current number of unused resource j
Allocation[i][j]	G_{ij}	process i's current allocation of j
Need[i][j]	N_{ij}	process i's potential for more j (Need[i][j]=Maximum[i][j]-Allocation[i][j])
Work[j]	W_j	a temporary storage (array) for Available[j]
Finish[i]	F_i	potential completeness of process i
Wait_count[i]	C_i	wait count for process i to break livelock

TABLE 1 DATA STRUCTURES FOR PBA

notation	explanation
p_i	a process
r_j	a resource
array[][] or array[]	all elements of the array
array[i][]	all elements of row i of the array
array[][j]	all elements of column j of the array

TABLE 2 NOTATIONS FOR PBA

Algorithm 1 Parallel Banker's Algorithm (PBA)

```
PBA (Request[i][] for resources from process $i$) {
1  STEP 0: $p_i$ makes Request[i][] for resources
2  STEP 1: if $\forall j$, Request[i][j] $\leq$ Need[i][j] /* $\forall$ means for all. */
3          goto STEP 2
4      else ERROR
5  STEP 2: if $\forall j$, Request[i][j] $\leq$ Available[j]
6          goto STEP 3
7      else deny $p_i$'s request, increase Wait_count[i] and return
8  STEP 3: pretend to allocate requested resources
9      $\forall j$, Available[j] := Available[j] – Request[i][j]
10     $\forall j$, Allocation[i][j] := Allocation[i][j] + Request[i][j]
11     $\forall j$, Need[i][j] := Maximum[i][j] – Allocation[i][j]
12 STEP 4: prepare for safety check
13     $\forall j$, Work[j] := Available[j]
14     $\forall i$, Finish[i] := false
   Let able-to-finish be ((Finish[i] == false) and ($\forall j$, Need[i][j] $\leq$ Work[j]))
15 STEP 5: Find all $i$ such that able-to-finish
16     if such $i$ exists,
17         $\forall j$, Work[j] := Work[j] + $\mathbf{\Sigma}_{i \text{ such that able-to-finish}}$ Allocation[i][j]
18         for $i$ such that able-to-finish, Finish[i] := true
19         repeat STEP 5
20     else (i.e., no such $i$ exists) goto STEP 6 (end of iteration)
21 STEP 6:
22     if Finish[i] == true for all $i$
23         then pretended allocations anchor; $p_i$ proceeds (i.e., SAFE)
24     else
25         restore the original state and deny $p_i$'s request (i.e., UNSAFE)
}
```

PBA takes as input the maximum requirements of each process and guarantees that the system always remains in a safe state. Tables (data structures or arrays) are maintained of available resources, maximum requirements, current allocations of resources and resources needed, as shown in Table 1. PBA uses these tables to determine whether the state of the system is either safe or unsafe. When resources are requested by a process, the tables are updated *pretending* the resources were allocated. If the tables will be in a safe state, then the request is actually granted; otherwise, the request is not granted, and the tables are returned to their previous states.

Let us explain Algorithm 1 step by step. A process can request multiple resources at a time as well as multiple instances of each resource. In Step 1, when a process requests resources, PBA first checks if the request does not exceed Need[i][] for the process. If the request is within its pre-declared claims, in Step 2 PBA checks if there are sufficient available resources for this request. If sufficient resources exist, PBA continues to Step 3; otherwise, the request is denied and the value of the wait counter (in variable Wait_count[i] of Table 1) for the process increases to break livelock if necessary. In Step 3, it is pretended that the request could be fulfilled, and the tables are temporarily modified according to the request.

In Step 4, PBA prepares for the safety check, i.e., initializes variables Work[] and Finish[]. In Step 5, PBA finds processes that can finish their jobs by acquiring some or all of available resources in Work[]. If one or more such processes exist, PBA adds all resources that these processes hold to Work[], then declares these processes to be *able-to-finish* (i.e., Finish[i] := true), and finally repeats Step 5 until all processes can finish their jobs. On the other hand, if no such process exists – meaning either all processes became *able-to-finish* or no more processes can satisfy the comparison (i.e., Need[i][j] $\leq$ Work[j] for all j) – PBA moves to Step 6 to decide whether the pretended allocation state is safe or not.

In Step 6, if all processes have been declared to be *able-to-finish*, then the granted allocation state is in a safe state (meaning there exists at least a safe sequence by which all processes can finish their jobs in the order of processes having been declared to be *able-to-finish*); thus, the requester can safely proceed. However, in Step 6, if there remain any processes unable to finish, the pretended allocation state may cause deadlock; thus, PBA denies the request, restores the original allocation state before the pretended allocation and also increases the wait count for the requester.

The gist of our approach is that because the operations in Step 5 are performed in parallel, if Need[i][j] $\leq$ Work[j] for all i and for all j are satisfied at the first iteration, PBA finishes at once, resulting in O(1) run-time. Such an example is given in Chapter V of [10] as "An example of resource allocation in a special case."

6 Implementation

Now we will describe implementation details including the architecture and circuitry of PBAU.

6.1 Architecture of PBAU

Figure 2 illustrates PBAU implemented in Verilog HDL. PBAU is composed of element cells, process cells, resource cells and a safety cell in addition to a Finite State Machine (FSM) and a processor interface.

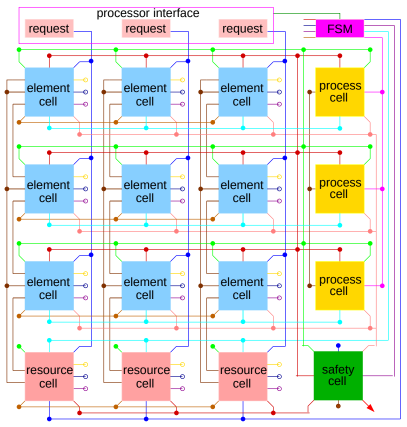

Figure 2 PBAU architecture

The Processor Interface (PI) consists of command registers and status registers. PI receives and interprets commands (requests or releases) from processes as well as accomplishes simple jobs such as setting up the numbers of maximum claims and available resources as well as adjusting the numbers of allocated and available resources in the response to a release of resources. PI also returns processing results back to PEs via status registers as well as activates the FSM in response to a request for resources from a process. In the next subsection, we will describe in detail two of the cells in Figure 2.

6.2 PBAU Circuitry

6.2.1 Element Cell

An Element Cell (EC), shown in Figure 3, performs two comparisons: Request[i][j] $\leq$ Need[i][j] and Need[i][j] $\leq$ Work[j]. The former comparison result (i.e., Request[i][j] $\leq$ Need[i][j]) is stored into a one-bit register. EC also stores Allocation[i][j] and Maximum[i][j]. EC emits Allocation[i][j] to Work[j] through *freed_out_ij* if the EC belongs to an *able-to-finish* process (i.e., Need[i][j] $\leq$ Work[j] for all j). In addition, there are two muxes, two subtracters and two adders. One adder is used to increase the number of allocation instances of the requested resource, and one subtracter is used to restore the temporarily increased number of instances if the safety test fails. Another subtracter is used to calculate the equation Need[i][j] = Maximum[i][j] – Allocation[i][j]. The other adder is used to make allocated instances (to this cell) available to later processes in a safe sequence.

6.2.2 Resource Cell

Each Resource Cell (RC, shown in Figure 4) corresponds to a multiple instance resource. RC has an Available[j] register that stores the number of instances of the resource. RC also has a Work[j] register that temporarily stores the number of resources in Available[j] (as shown in Step 4) plus resources to be released by *able-to-finish* processes during iterations of

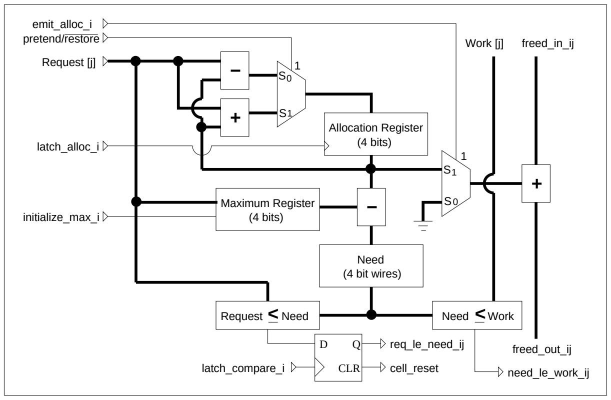

Figure 3 Logic diagram of an Element Cell (EC)

Step 5. RC also has a comparator that compares Request[i][j] with Available[j], the result of which is stored into a register.

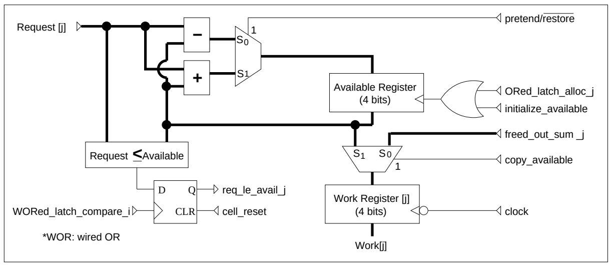

Figure 4 Logic diagram of a Resource Cell (RC)

The rest of the cells, i.e., Process Cell and Safety Cell, have been designed in parallel hardware using the same approach as described for Element Cell. The details of Process Cell and Safety Cell are described in a thesis [10].

6.2.3 Finite State Machine (FSM)

Figure 5 illustrates the transition diagram of the FSM along with input and output signals, which mainly controls the behavior of all cells. The details of FSM operation at each clock cycle are contained in [10].

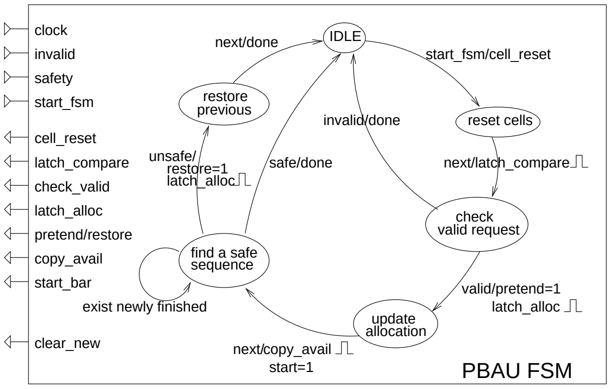

Figure 5 Finite state machine

6.3 Synthesized Result of PBAU

We used Synopsys Design Compiler [11] to synthesize various types of the PBAU with the QualCore Logic .25μm standard cell library [12]. We aimed to synthesize at a clock period of 4 ns. The synthesis result is shown in Table 3. The "Area" column denotes the area in units equivalent to a minimum-sized two-input NAND gate in the library. PBAU5x5 represents a PBAU for five processes and five resources (each of which can have up to 16 instances). In case where an SoC contains five PowerPC 755 PEs (1.7M gates each) and a 16MB memory (33.5M gates), the area overhead in the SoC due to PBAU 20x20 is less than .05%.

Synthesis Result	PBAU5x5	8x8	10x10	15x15	20x20
Area (w.r.t. 2-input NAND)	1303	3243	5030	11158	19753
Number of lines of Verilog	600	700	770	1000	1350

TABLE 3 SYNTHESIZED RESULT OF PBAU

6.4 Run-time Complexity of PBAU

The run-time complexity of a generic implementation of the traditional BA in software is $O(m \times n^2)$, where m and n are the numbers of resources and processes, respectively [5].

By implementing PBA in hardware able to exploit full parallelism, we achieved a run-time of O(1) in the best-case (i.e., the cases of system states where for all i and for all j, Need[i][j] $\leq$ Available[j]), O(n) in the worst-case (i.e., the cases where there exists only one unique safe sequence of one by one increment order of *able-to-finish* processes), and approximately n/2 clock cycles on average. A more formal and detailed theoretical analysis is contained in [10].

7 Experiments

7.1 Simulation Environment Setup

The experimental simulations were carried out using Seamless Co-Verification Environment (CVE) [13] aided by Synopsys VCS [14] for Verilog HDL simulation and XRAY [15] for software debugging. We have used Atalanta Real-Time Operating System (RTOS) version 0.3 [16], a shared-memory multiprocessor RTOS. The RTOS code resides in a shared memory, and all PEs execute the same RTOS code and share kernel structures and the states of all processes and resources.

7.2 Experimental System

For the experiment, we simulate an MPSoC with five Motorola MPC755s and resources similar to Figure 1. Each MPC755 has separate instruction and data L1 caches each of size 32KB. The MPSoC also has the following three types of resources: an SoCDMMU [6] with 10 blocks of allocable memory (r_1), a counting semaphore with a group of five DSP processors (r_2) and another counting semaphore with seven I/O buffers (r_3). These three types of resources have timers, interrupt generators and input/output ports as needed to operate properly in the MPSoC. In addition, the MPSoC has a PBAU for five processes and five resources, an arbiter and 16MB of shared memory including the allocable memory. The master clock rate 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]. Everything else other than the MPC755s are described in Verilog

HDL and simulated in Synopsys VCS [14]. We invoke processes $p_1, \cdots, p_5$ on PE1, $\cdots$, PE5, respectively.

7.3 Application Example

We execute a sample robotic application in which recognizing objects, avoiding obstacles and displaying trajectory require DSP processing; robot motion and data recording involve accessing IO buffers; and proper real-time operation (e.g., maintaining balance) of the robot demands fast and deterministic allocation and deallocation of memory blocks. This application invokes a sequence of requests and releases. The sequence has ten requests, six releases and five claim settings with one false request (e.g., Request[i][j] $>$ Need[i][j]) and one request that leads to an unsafe state. Note that every command is processed by an avoidance algorithm (either PBAU or BA in software). Recall that there is no constraint on the ordering of the resource usage.

In an experiment with the application, we measure two figures, the average execution time of deadlock avoidance algorithms and the total execution time of the application in two cases: (i) using PBAU versus (ii) using the Banker's Algorithm in software. The application example is described in great detail in a thesis [10].

7.4 Experimental Result

Table 4 shows that PBAU achieves about a 1600X speedup of the average algorithm execution time and gives a 19% speedup of application execution time over avoiding deadlock with BA in software (the speedup is calculated according to the formula by Hennessy and Patterson [17]). Note that during the run-time of the application, each avoidance method (PBAU or BA in software) is invoked 22 times in both cases, respectively (since every request and release invokes each method). Table 5 represents the average algorithm execution time distribution in terms of different types of commands.

In short summary, while BA in software spends about 5400 clock cycles on average at each invocation in this experiment, PBAU only spends 3.32 clocks on average.

Method of Implementation	Algorithm Exec. Time	PBAU Speedup	Application Exec. Time	Application Speedup
PBAU (hardware)	3.32	$\frac{5398.4-3.32}{3.32} = 1625X$	185716	$\frac{221259-185716}{185716} = 19\%$
BA in software	5398.4		221259	

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

TABLE 4 EXECUTION TIME COMPARISON

Method of Implementation	Set Available	Set Max Claim	Request Command	Release Command	Wrong Command
# of commands	1	5	9	6	1
PBAU (hardware)	1	1	6.5	1	2
BA in software	416	427	11337	2270	560

*The time unit is a clock cycle, and the values are averaged if there were multiple commands of the same type. "#" denotes "the number of".

TABLE 5 EXECUTION TIME COMPARISON

8 Conclusion

A novel Parallel Banker's Algorithm (PBA) for multiple-instance multiple-resource systems and its hardware implementation, which we call PBA Unit (PBAU), are described in this paper. PBAU gives an O(n) run-time complexity with the best-case of O(1); the result seems to be an average run-time of approximately n/2 clock cycles in most cases. PBAU provides a multiprocessor system with a very fast and low area way of avoiding deadlock at run-time, which helps free programmers from worrying about deadlock. Whenever a request occurs in a system, PBAU checks for the safety of its grant. The request is granted provided that the system can remain in a safe state.

We demonstrated the following through an experiment. (i) PBAU automatically avoided deadlocks as well as reduced the deadlock avoidance time by 99.9% *(about 1600X)* as compared to the Banker's Algorithm (BA) in software. (ii) PBAU achieved in a particular example a 19% speedup of application execution time in an experiment as compared to the execution time of the same application that uses BA in software.

Finally, the MPSoC area overhead due to PBAU is small, under 0.05% in our candidate MPSoC example.

Acknowledgements

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

References

[1] Xilinx, http://www.xilinx.com/.

[2] DVT solutions, http://www.xilinx.com/esp/dvt/cdv/dvt_solutions/ip/dsp/.

[3] E. Dijkstra, "Cooperating sequential processes," Tech. Rep. EWD-123, Technological University, Eindhoven, The Netherlands, Sep. 1965.

[4] A. Habermann, "Prevention of system deadlocks," *Communications of the ACM*, 12(7) pp. 373–377,385, July 1969.

[5] A. Silberschatz and P. Galvin, *Operating System Concepts*, John Wiley & Sons, Inc., New York, NY, 1999.

[6] M. Shalan and V. Mooney, "Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management," 10th International Symposium on Hardware/Software Codesign (CODES'02), pp. 79-84, May 2002.

[7] R. Holt, "Some deadlock properties of computer systems," *ACM Computing surveys*, pp. 179–196, Sep. 1972.

[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] J. Lee and V. Mooney, "A novel deadlock avoidance algorithm and its hardware implementation," CODESISSS 2004, pp 200-205, Sep. 2004.

[10] J. Lee, "Hardware/software deadlock avoidance for multiprocessor multiresource system-on-a-chip," Ph.D. Thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, USA, Fall 2004.

[11] Design Compiler, http://www.synopsys.com/products/logic/logic.html.

[12] QualCore Logic, http://www.qualcorelogic.com/.

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

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

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

[16] 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.

[17] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann Publisher, Inc., San Francisco, CA, 1996.