Hardware/Software Partitioning of Operating Systems: Focus on Deadlock Detection and Avoidance

Jaehwan John Lee[+] and Vincent John Mooney III[*]
[*,+]Center for Research on Embedded Systems and Technology
[*]Associate Professor, [+]School of Electrical and Computer Engineering
[*]Adjunct Associate Professor, College of Computing
Georgia Institute of Technology, Atlanta, Georgia, U.S.A.
{jaehwan, mooney}@ece.gatech.edu

Abstract

As MultiProcessor System-on-a-Chip (MPSoC) designs become more common, hardware/software codesign engineers face new challenges involving operating system integration. To speed up operating system/MPSoC codesign, this article presents recent research in hardware/software partitioning of a Real-Time Operating System (RTOS). After a brief overview of the δ hardware/software RTOS design framework, we focus on new results in deadlock detection and avoidance. Among various configured RTOS/MPSoC designs in this research, we show an example where a system with the Deadlock Detection hardware Unit (DDU) achieves a 46% speed-up of application execution time over a system with deadlock detection in software. Similarly, we show another example where a system with the Deadlock Avoidance hardware Unit (DAU) not only automatically avoids deadlock but also achieves a 44% speed-up of application execution time over a system avoiding deadlock in software; furthermore, in our example, the DAU only consumes .005% of the MPSoC total chip area.

Keywords
Hardware/Software Partitioning, Real-Time Operating System, Deadlock Detection, Deadlock Avoidance

1 Introduction

Primitive operating systems were first introduced in the 1960s in order to relieve programmers of common tasks such as those involving Input/Output (I/O). Gradually, scheduling and management of multiple jobs/programs became the purview of an Operating System (OS). Many fundamental advances, such as multithreading and multiprocessor support, have propelled both large companies and small into the forefront of software design.

Recent trends in chip design press the need for more advanced operating systems for System-on-a-Chip (SoC). However, unlike earlier trends where the focus was on scientific computing, today's SoC designs tend to be driven more by the needs of embedded computing. While it is hard to state exactly what constitutes embedded computing, it is safe to say that the needs of embedded computing form a superset of scientific computing. For example, real-time behavior is critical in many embedded platforms due to close interaction with non-humans, e.g., rapidly moving mechanical parts. In fact, the Application-Specific Integrated Circuits (ASICs) preceding SoC did not integrate multiple processors with custom hardware, but instead were almost exclusively digital logic specialized to a particular task and hence very timing predictable and exact. Therefore, we predict that advances in operating systems for SoC focusing on Real-Time Operating System (RTOS) design provide a more natural evolution for chip design as well as being compatible with real-time systems.

Furthermore, thanks to the recent trends in the technologies of MultiProcessor System-on-a-Chip (MPSoC) and reconfigurable chips, many hardware Intellectual Property (IP) cores that implement software algorithms have been developed to speed up computation and utilize low cost hardware. However, fully exploiting these innovative hardware IP cores have had many difficulties such as interfacing IP cores to a specific system, modifying IP cores to fulfill requirements of a system under consideration, porting device drivers and finally integrating both IP cores and software seamlessly. Much work of interfacing, modifying and/or porting IP cores and device drivers has relied on human resources. Hardware/software codesign frameworks can help reduce the burden on designers.

This article focuses on such research in the design of operating systems, especially RTOSes. We have implemented and upgraded the δ hardware/software RTOS/MPSoC design framework (shown in Figure 1). Since we have already described our approach in [1, 2, 3, 4], in this article we first briefly explain the δ framework and then focus more on an exposition of deadlock issues. We believe deadlock issues are on the horizon due to the rapid evolution in MPSoC technology and the introduction of many innovative IP cores. We predict that future MPSoC designs will have hundreds of processors and

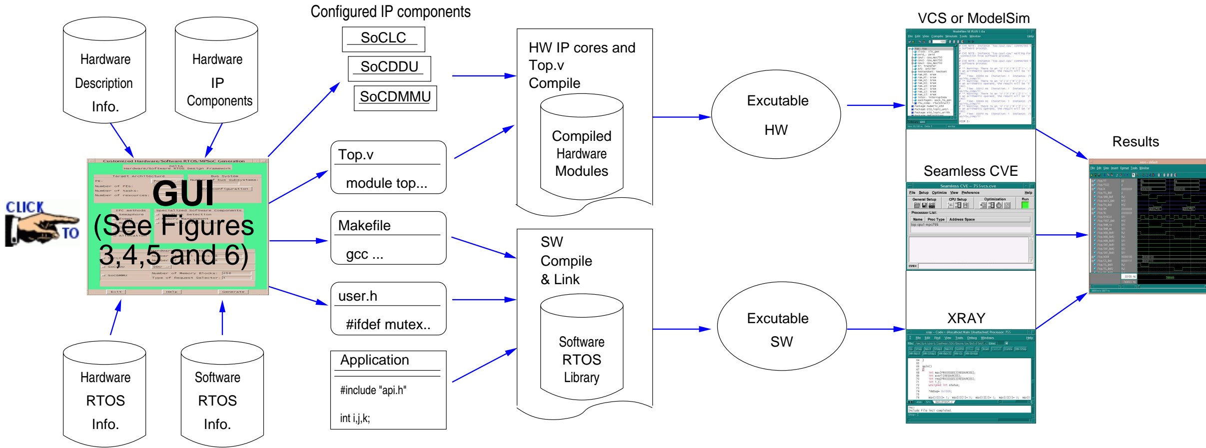


Figure 1 The δ hardware/software RTOS design framework.

resources (such as custom FFT hardware) all in a single chip; thus, systems will handle much more functionality, enabling a much higher level of concurrency and requiring many more deadlines to be satisfied. As a result, we predict there will be resource sharing problems among the many processors desiring the resources, which may result in deadlock more often than designers might realize.

The remainder of this article is organized as follows. Section 2 first presents our target MPSoC architecture and then explains the δ hardware/software RTOS design framework version 2.0 including a description of two hardware RTOS components: a "lock" cache and a dynamic memory allocator. Section 3 motivates deadlock issues and provides background about deadlock problems. Section 4 focuses on new software/hardware solutions to such deadlock problems. Section 5 addresses experimental setup and shows various comparison results with applications that demonstrate how the δ framework could impact hardware/software partitioning in current and future RTOS/MPSoC designs. Finally, Section 6 addresses conclusions.

2 Hardware/software RTOS design

2.1 RTOS/MPSoC target

Figure 2 shows our primary target MPSoC consisting of multiple processing elements with L1 caches, a large L2 memory, and multiple hardware IP components with essential interfaces such as a memory controller, an arbiter and a bus system. The target also has a shared memory multiprocessor RTOS (Atalanta [5] developed at the Georgia Institute of Technology), which is small and configurable. The code of Atalanta RTOS version 0.3 resides in shared memory, and all processing elements (PEs) execute the same RTOS code and share kernel structures as well as the states of all processes and resources. Atalanta supports priority scheduling with priority inheritance as well as round-robin; task management such as task creation, suspension and resumption; various Inter Process Communication (IPC) primitives such as semaphores, mutexes, mailboxes, queues and events; memory management; and interrupts. As shown in Figure 2, hardware IP cores can be either integrated into the reconfigurable logic or implemented as custom logic. Besides, specialized IP cores such as DSP processors and wireless interface cores can also be integrated into the chip.

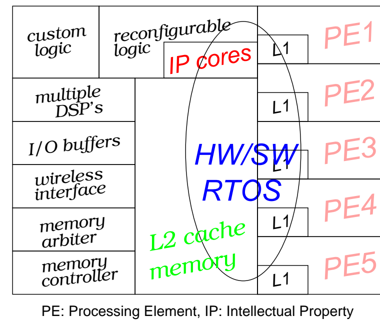


Figure 2 Future MPSoC.

2.2 The δ Framework

The δ hardware/software RTOS generation framework for MPSoC (shown in Figure 1) was proposed to enable automatic generation of different mixes of predesigned hardware/software RTOS components that fit the target MPSoC the user is designing so that RTOS/MPSoC designers can decide their critical decisions earlier in the design phase of their target product(s) [1, 2, 3, 4]. Thus, the δ framework helps the user explore which configuration is most suitable for the user's target and application or set of applications. In other words, the δ framework is specifically designed to provide a solution to rapid RTOS/MPSoC (both hardware and software) design space exploration so that the user can easily and quickly find a few optimal RTOS/MPSoC architectures that are most suitable to his or her design goals. The δ framework generates a configured RTOS/MPSoC design that is simulatable on a hardware/software cosimulation environment after the generated design is compiled. Hardware designs are described in a

Hardware Description Language (HDL) such as Verilog. Software designs could be described in any language although we have only used C in our designs.

From the initial implementation [1, 2, 3, 4], we have extended the δ framework to include parameterized generators of hardware IP components (i.e., automatically configurable to fit a desired target architecture) as well as the generation of various types of bus systems. This section gives an overview of parameterized generators for a customized RTOS/MPSoC design including a bus configurator, a dynamic memory management unit generator and a custom "lock" cache generator, and explains such available IP components briefly. Many low-level details – e.g., details of the bus system generation – are not repeated in this article but instead are available in referenced works.

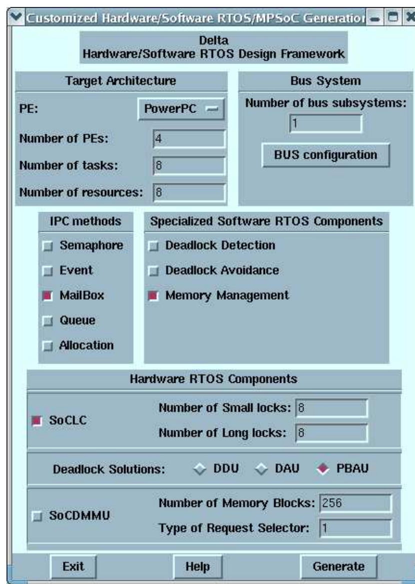


Figure 3 GUI of the δ framework.

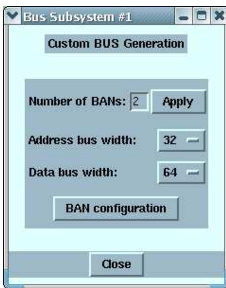


Figure 4 Bus system configuration.

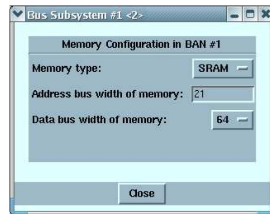


Figure 5 Bus subsystem memory configuration.

Figure 3 shows graphical user interface for the δ framework version 2.0, which now integrates four parameterized generators we have and generates an RTOS/MPSoC system.

Here we summarize each generator briefly. For more information, please see specific references. When a user wants to create his or her own specific bus system, the user clicks "Bus configuration" (shown at the top right of Figure 3), which brings up a pop-up window (shown in Figure 4), in which the user specifies address and data bus widths as well as detailed bus topology for each subsystem in subsequent windows (shown in Figures 5 and 6) for a system with a hierarchical bus structure. After the appropriate inputs are entered, the tool will generate a user specified bus system with the specified hierarchy. Further details about bus system generation are described in [7, 8, 9].

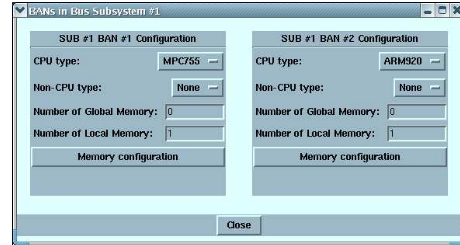


Figure 6 Bus subsystem configuration.

At the bottom of Figure 3, there are several options for "Hardware RTOS Components": the SoC Lock Cache (SoCLC), multiple deadlock detection/avoidance solutions, and the SoC Dynamic Memory Management Unit (SoCDMMU). The details of these hardware RTOS components will be described in Sections 2.3, 3 and 4.

In addition to selecting hardware RTOS components, the δ framework version 2.0 can also manipulate the size and type of each RTOS component by use of input parameters. For instance, when the user wants to include SoCLC, he or she can also specify the number of small locks and the number of long locks (equivalent to semaphores) according to the expected requirements for his or her specific target (or goal). Detailed parameterized SoCLC generation is discussed in [10, 11].

For the SoCDMMU IP component, the user can specify the number of memory blocks (available for dynamic allocation in the system) and other parameters, and then the GUI tool will generate a user specified SoCDMMU. Details regarding parameterized SoCDMMU generation are addressed in [12, 13].

We briefly describe our new approach to an HDL top file generation process in the following example.

Example 1 As shown in Figure 7, the GUI tool generates a Verilog top file according to the description of a user specified system with hardware IP components. For instance, a user selects a system having three PEs and an SoCLC for eight small locks and eight long locks. The generation process starts with a description of a system having an SoCLC (i.e., LockCache description) in the description library. The LockCache description lists modules necessary to build a system containing an SoCLC, such as PEs, L2 memory, a memory controller, a bus arbiter, an interrupt controller and an SoCLC. The Verilog top file generator, which we call *Archi_gen*, writes all instantiation code for each module in the list of the LockCache description to a file. *Archi_gen* also includes multiple instantiation code of the same type IP with distinct identification numbers since some modules such as PEs need to be instantiated multiple times. Then, *Archi_gen* writes necessary wires described in the LockCache description, and then writes initialization routines necessary to execute simulation. Later by compiling *Top.v*, a specified target hardware architecture will be ready for exploration [2]. ■

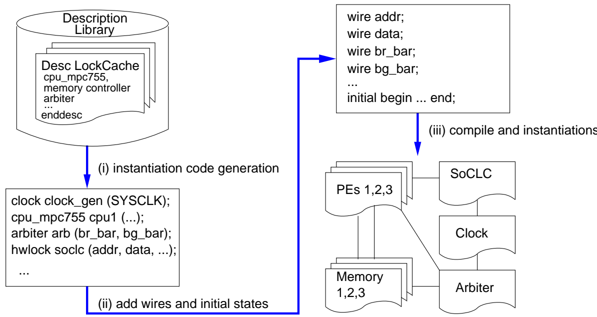


Figure 7 Top file generation of the δ framework.

2.3 Hardware RTOS components

This subsection briefly mentions two available hardware IP components presented previously: SoCLC and SoCDMMU.

2.3.1 SoCLC

Synchronization has always been a critical issue in multiprocessor systems. As multiprocessors execute a multitasking application on top of an RTOS, any important shared data structure, also called a Critical Section (CS), may be accessed for inter-process communication and synchronization events occurring among the tasks/processors in the system.

Previous work has shown that the System-on-a-Chip Lock Cache (SoCLC [11, 14, 15]), which is a specialized custom hardware unit realizing effective lock-based synchronization for a multiprocessor shared-memory SoC as shown in Figure 8, reduces on-chip memory traffic, provides a fair and fast lock hand-off, simplifies software, increases the real-time predictability of the system and improves performance as well.

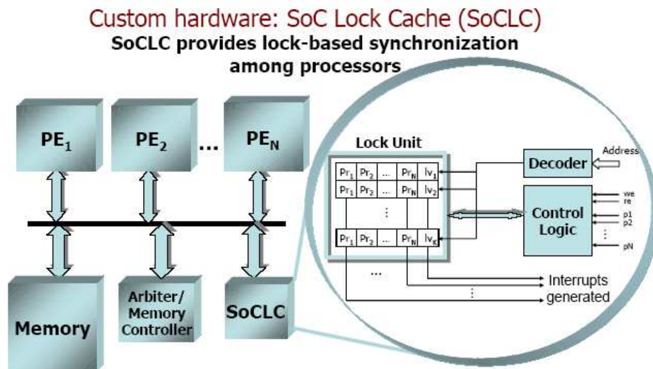


Figure 8 SoCLC.

Akgul et al. extended the SoCLC mechanism with a priority inheritance support implemented in hardware [16]. Priority inheritance provides a higher level of real-time guarantees for synchronizing application tasks. The authors present a solution to the priority inversion problem in the context of an MPSoC by integrating an immediate priority ceiling protocol (IPCP) [17] implemented in hardware. The approach also provides higher performance and better predictability for real-time applications running on an MPSoC.

Experimental results indicate that the SoCLC hardware mechanism with priority inheritance achieves a 75% speed-up in lock delay, a 79% speed-up in lock latency [16]. The cost in terms of additional hardware area for the SoCLC with priority inheritance is approximately 10,000 NAND2 gates (in TSMC $.25\mu$ chip fabrication technology).

2.3.2 SoCDMMU

The System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) shown in Figure 9 is a hardware unit that allows a fast and deterministic way to dynamically allocate/deallocate global (L2) memory among PEs [18]. The SoCDMMU is able to convert the PE address (virtual address) to a physical address. The memory mapped address or I/O port to which the SoCDMMU is mapped is used to send commands to the SoCDMMU (writing data to the port or memory-mapped location) and to receive the results of the command execution (reading from the port or memory-mapped location).

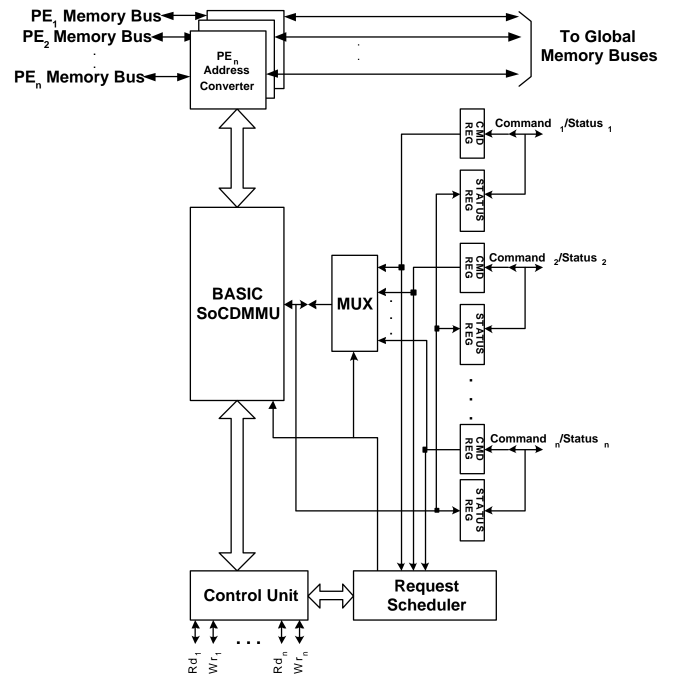


Figure 9 SoCDMMU.

As shown in [13, 18], the SoCDMMU achieves a 4.4X overall speed-up in memory management during the application transition time when compared to conventional memory allocation/deallocation techniques, i.e., *malloc()* and *free()*. The SoCDMMU is synthesizable and has been integrated into a system example including porting SoCDMMU functionality to an RTOS (so that the user can access SoCDMMU functional-

ity using standard software memory management APIs) [18]. Also, the SoCDMMU-crossbar (Xbar) switch Generator (DX-Gt [12]) can configure and optimize the SoCDMMU to suit a specific system (e.g., for a particular memory configuration and number of PEs). In this way, DX-Gt automates the customization and the generation of the hardware memory management functionalities.

3 Background and prior work for deadlock

In this section we motivate the development of deadlock related software and hardware IP components and then introduce definitions, assumptions and prior work related to our deadlock research.

3.1 Motivation for the design of deadlock related hardware components

In most current embedded systems in use today, deadlock is not a critical issue due to the use of only a few (e.g., two or less) 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 (or more) processors and ten to a hundred resources all in a single chip. This is the way we predict 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 [19]. Given current technology trends, we predict that MPSoC designers and users are going to start facing deadlock problems more and more often. That is, deadlock problems are on the horizon.

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.

We currently have a couple of solutions, the Deadlock Detection Unit (DDU) and the Deadlock Avoidance Unit (DAU), that 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 a 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.

3.2 Background

3.2.1 Definitions

Definitions of deadlock, livelock and avoidance 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 (e.g., preempted), waiting for requirements that can never be satisfied.*

Definition 2 *Livelock is a situation where a request for a resource is repeatedly denied and possibly never accepted because of the unavailability of the resource, resulting in a stalled process, while the resource is made available for other process(es) which make progress.*

Definition 3 *Deadlock Avoidance is a way of dealing with deadlock where resource usage is dynamically controlled not to reach deadlock (i.e., on the fly, resource usage is controlled to ensure that there can never be deadlock).*

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

Definition 4 *For a given system, if a request from a process directly causes the system to have a deadlock at that moment, then we denote this case as* ***request deadlock*** *or* ***R-dl****.*

A request deadlock (R-dl) example is described in Section 5.4.3.

Definition 5 *For a given system, if the grant of a resource to a process directly causes the system to have a deadlock at that moment, then we denote this case as* ***grant deadlock*** *or* ***G-dl****.*

A grant deadlock (G-dl) example is described in Section 5.4.1. Please note that we differentiate between R-dl and G-dl because our deadlock avoidance algorithm in Section 4.3.1 requires the distinction to be made. The distinction is required because some actions can only be taken for either R-dl or G-dl; e.g., for G-dl it turns out that perhaps deadlock can be avoided by granting the released resource to a lower priority process.

3.2.2 System model in view of deadlock

To address deadlock issues, we first show a modified MPSoC from Figure 2 in the following example.

Example 2 A future Request-Grant MPSoC
We introduce the device shown in Figure 10 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 10(b). The MPSoC also contains memory, a memory controller and a DAU. In the figure, we assume that each PE has only one active process; i.e., each process p_1, p_2, p_3 and p_4, shown in Figure 10(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 10 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. ■

We consider this kind of request-grant system with many resources and processes shown in Figure 10 as our system model in view of deadlock. Based on our system model, we now introduce some underlying assumptions related to our deadlock research in such MPSoCs.

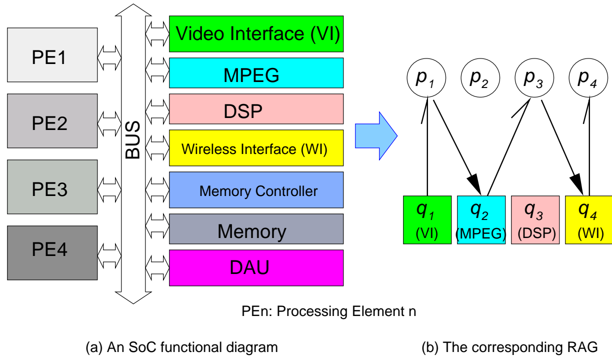


Figure 10 A practical MPSoC realization.

3.2.3 Assumptions

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

Assumption 2 *A resource can be released only by the process holding it.*

Assumption 3 *The RTOS or other software provides a mechanism that can ask a process to release any resource(s) the process currently holds.*

3.3 Prior work in deadlock research

3.3.1 Overview of prior deadlock research

Researchers have put tremendous efforts into deadlock research, three well-known areas of which are deadlock detection, prevention and avoidance [20, 21, 24, 31]. Among them, deadlock detection provides more freedom for a system since deadlock detection does not typically restrict the behavior of a system, facilitating full concurrency. Deadlock detection, however, usually requires a recovery once a deadlock is detected. In contrast, deadlock prevention prevents a system from reaching deadlock by typically restraining request orders to resources in advance, implying restrictions on concurrency. One such method is the priority ceiling protocol (PCP [17]), which is only a solution for a single processor system, though. Another method is the collective request method, which however tends to cause resource under-utilization as well as process starvation. Deadlock avoidance, by contrast, generally sits in-between; that is, deadlock avoidance normally gives more freedom with less restrictions than deadlock prevention [31]. Deadlock avoidance essentially requires knowledge about the maximum necessary resource requirements for all processes in a system, which unfortunately makes the implementation of deadlock avoidance difficult in real systems with dynamic workloads.

3.3.2 Deadlock detection

All software deadlock detection algorithms known to the authors to date have a run-time complexity of at least $O(m \times n)$, where m is the number of resources and n is the number of processes. In 1970, Shoshani et al. proposed an $O(m \times n^2)$ run-time complexity detection algorithm [20], and about two years later, Holt proposed an $O(m \times n)$ algorithm to detect a knot that tells whether deadlock exists or not [21]. Both of the aforementioned algorithms (of Shoshani et al. and of Holt) are based on a Resource Allocation Graph (RAG) representation. Leibfried proposed a method of describing a system state using an adjacency matrix representation and a corresponding scheme that detects deadlock with matrix multiplications but with a run-time complexity of $O(m^3)$ [22]. Kim and Koh proposed an algorithm with $O(m \times n)$ time for "detection preparation"; thus an overall time for detecting deadlock (starting from a system description that just came into existence, e.g., due to multiple grants and requests occurring within a particular time or clock cycle) of at least $O(m \times n)$ [23].

3.3.3 Deadlock avoidance

A traditional well-known deadlock avoidance algorithm is the Banker's algorithm [24]. 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 and 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 tends to restrict resource utilization, which may degrade normal system performance, and (iii) the maximum resource requirements (and thus requests) might not be known in advance [24, 25].

In 1990, Belik proposed a deadlock avoidance technique [26] 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 his deadlock avoidance algorithm.

4 New approaches to deadlock problems

In this section, we describe in detail deadlock related software and hardware IP components, i.e., a parallel deadlock detection algorithm, a deadlock avoidance algorithm, the Deadlock Detection hardware Unit (DDU) and the Deadlock Avoidance hardware Unit (DAU).

4.1 Introduction

All of the algorithms referenced in Section 3 assume an execution paradigm of one instruction or operation at a time. With a custom hardware implementation of a deadlock algorithm, however, parallelism can be exploited.

Detection of deadlock is extremely important since any request for or grant of a resource might result in deadlock. Invoking software deadlock detection on every resource allocation event would cost too much computational power; thus, using a software implementation of deadlock detection and/or avoidance would perhaps be impractical in terms of the performance cost. A promising way of solving deadlock problems with small compute power is to implement deadlock detection and/or avoidance in hardware.

To handle this possibility, Parallel Deadlock Detection Algorithm (PDDA) and its hardware implementation (DDU) have been proposed [27]. Utilizing the DDU, a novel Deadlock Avoidance Algorithm and its hardware implementation (DAU) have recently been proposed [28].

The DDU has been proven to have a run-time complexity of $O(min(m,n))$ using custom hardware [29]. The DDU manipulates a simple boolean representation of the types of each edge: the request edge of a process requesting a resource, the grant edge of a resource granted to a process, or no activity (neither a request nor a grant) [29]. Not only that, but by implementing PDDA with a small amount of hardware, the designed deadlock detection unit hardly affects system performance (and potentially has no negative impact whatsoever) yet provides the basis for an enhanced deadlock detection methodology.

The disadvantages (i), (ii) and (iii) mentioned in Section 3.3.3 unfortunately make 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.

The DAU avoids deadlock by not allowing any grant or request that leads to a deadlock. In case of livelock resulting from attempts to avoid deadlock, the DAU asks one of the processes involved in the livelock to release resource(s) so that the livelock can also be resolved.

Although many deadlock avoidance approaches have been introduced so far [24, 25, 26, 30], to the best of our knowledge, there has been no prior work in a hardware implementation of deadlock avoidance. The DAU 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 5).

In the next few sections, we will further describe these new approaches in more detail.

4.2 New deadlock detection methodology

4.2.1 Parallel Deadlock Detection Algorithm

Parallel Deadlock Detection Algorithm (PDDA) dramatically reduces deadlock detection time by mapping a Resource Allocation Graph (RAG [31], its state is denoted as γ_{ij} [29]) into a matrix M_{ij} that will have exactly the same request and grant edges as the RAG has but with another notation for each edge. We define a RAG matrix and a terminal reduction sequence before introducing PDDA that exploits the terminal reduction sequence.

Definition 6 *The purpose of this definition is to define matrices that correspond to graph γ, system γ_i and state γ_{ij} [29]. A **RAG matrix M** is a matrix mapped from a RAG γ and represents an arbitrary system with processes and resources. A **system matrix M_i** is defined as a matrix representation of a particular system γ_i, where the rows (fixed in size) of matrix M_i represent the fixed set Q of resource nodes of γ_i, and the columns (fixed in size) of matrix M_i represent the fixed set P of process nodes of γ_i. We denote another notation of this relationship as $M_i \equiv \gamma_i$ for the sake of simplicity. A **state matrix M_{ij}** is a matrix that represents a particular system state γ_{ij}, i.e., $M_{ij} \equiv \gamma_{ij}$. Edges E (consisting of request edges R and grant edges G [29]) in system state γ_{ij} are mapped into the corresponding array elements using the following rule:*

Given $E = \{R \cup G\}$ from γ_{ij},

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

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

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

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

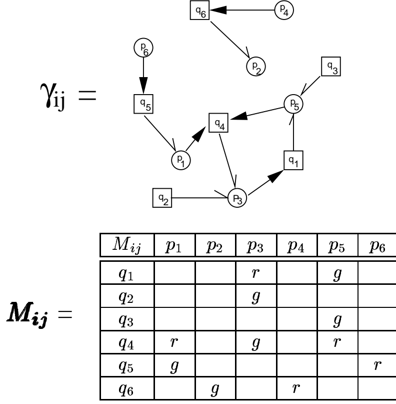
$\alpha_{st} = 0_{st}$ ('0' or a blank space), otherwise.

where q_s and p_t represent a process and a resource, respectively.

Example 3 State Matrix Representation
The system state γ_{ij} shown on the upper half of Figure 11 can be represented in the matrix form shown on the bottom half of Figure 11.

■

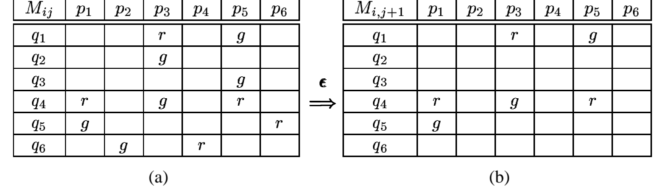
Based on a state matrix M_{ij}, instead of finding an exact cycle (as other algorithms do, e.g., see Chapter 4 of [31]), PDDA removes edges that have nothing to do with cycles; this edge removal process is called a terminal reduction sequence. After the terminal reduction sequence (e.g., using k edge removal steps) removes all reducible edges (resulting in an "irreducible" matrix $M_{i,j+k}$), if edges still exist, then deadlock(s) exist. On the other hand, if M_{ij} has been completely reduced, no deadlock exists. Intuitively, removing reducible edges corresponds to the best sequence of operations a particular process



M_{ij} =

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

Figure 11 Matrix representation example



(a)

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

$\stackrel{\epsilon}{\Longrightarrow}$

(b)

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

Figure 12 One terminal reduction step (ϵ) example

can execute to help unblock other processes. Before describing the terminal reduction sequence in detail, we define what we mean by "terminal" in different uses.

Definition 7 *A **terminal row** is a row r_s (which corresponds to resource q_s) of matrix M_{ij} such that either (i) all non-zero entries $\{\alpha_{st_r} \neq 0, 1 \leq t_r \leq n\}$ are request entries $r_{t_r \to s}$ with at least one request entry (i.e., one or more request entries and no grant entry in the row) or (ii) one entry α_{st_g}, $1 \leq t_g \leq n$, is a grant $g_{s \to t_g}$ with the rest of the entries $\{\alpha_{st}$, $1 \leq t \leq n$, $t \neq t_g\}$ equal to zero.*

Definition 8 *A **terminal column** is a column c_t (which corresponds to process p_t) of matrix M_{ij} such that either (i) all non-zero entries $\{\alpha_{st} \neq 0,\ 1 \leq s \leq m\}$ are request entries with at least one request entry (i.e., one or more request entries and no grant entry in the column) or (ii) all non-zero entries $\{\alpha_{st} \neq 0,\ 1 \leq s \leq m\}$ are grant entries with at least one grant entry.*

Definition 9 *Given state matrix M_{ij}, function $T_r(M_{ij})$ produces the on-set (i.e., true set) of all terminal rows.*

Definition 10 *Given state matrix M_{ij}, function $T_c(M_{ij})$ produces the on-set of all terminal columns.*

Definition 11 *An edge that belongs to either a terminal row or a terminal column is called a **terminal edge**.*

The next definition defines one step of a terminal reduction sequence.

Definition 12 ***A terminal reduction step ϵ** is a unary operator $\epsilon : M_{ij} \mapsto M_{i,j+1}$ (i.e., $M_{i,j+1} = \epsilon(M_{ij})$), where ϵ calculates all terminal edges and returns $M_{i,j+1}$ such that all the terminal edges found are removed by setting the terminal entries found to zero; thus, the next iteration $M_{i,j+1}$ will start with equal or fewer total edges as compared to M_{ij}.*

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

Example 4 One Step of Terminal Reduction (ϵ)

Figure 12 (b) shows a new matrix $M_{i,j+1}$ after a matrix reduction step **ϵ**, defined in Definition 12, is applied to M_{ij} shown in (a). In matrix M_{ij}, since q_2 and q_3 are terminal rows by Definition 7, all the edges in their rows are terminal edges.Therefore, all the edges in rows q_2 and q_3 can be removed. Likewise, p_2, p_4 and p_6 are terminal columns by Definition 8; hence, all edges in these columns can be removed, resulting in matrix $M_{i,j+1}$. ■

Definition 13 ***A terminal reduction sequence ξ**, applicable to a matrix M_{ij}, is a sequence of k terminal reduction steps ϵ (recall that ϵ is a terminal reduction step) such that (i) $M_{ij} \mapsto M_{i,j+1} \mapsto \cdots \mapsto M_{i,j+k}$; (ii) $M_{i,j+k}$ is irreducible (i.e., $\epsilon(M_{i,j+k}) = M_{i,j+k}$); and (iii) $\{M_{i,j+h}, 0 \leq h < k\}$ are all unique and reducible. A terminal reduction sequence is called a **complete reduction** when the sequence of terminal reduction steps corresponding to ξ results in $M_{i,j+k}$ such that the irreducible state matrix $M_{i,j+k}$ contains all zero entries (i.e., no edges). A terminal reduction sequence is called an **incomplete reduction** when ξ returns $M_{i,j+k}$ with at least one non-zero entry (i.e., at least one edge).*

We now introduce two algorithms, one being a terminal reduction sequence algorithm that implements the terminal reduction sequence ξ, the other being PDDA, which employs the terminal reduction sequence algorithm.

Algorithm 1 is an implementation of the terminal reduction sequence **ξ** shown in Definition 13. We summarize the operation of Algorithm 1. Lines 2 and 3 of Algorithm 1 initialize two variables: iterator k and matrix M_{iter} that is initially a copy of input matrix M_{ij}. Line 5 finds all terminal rows, and line 6 finds all terminal columns. Line 7 checks whether M_{iter} has more terminal edges, and, if no more terminal edges exist, the current iteration ends. Lines 8 and 9 remove all terminal edges found at the current iteration. On the whole, the terminal reduction step $\epsilon(M_{ij})$ of Definition 12 corresponds to lines 5-9 of Algorithm 1, which iterates until the matrix M_{iter} becomes irreducible; this iteration process implements the terminal reduction sequence **ξ**. Note that, in hardware implementation, lines 5 and 6 of Algorithm 1 are executed at the same time in parallel, as are lines 8 and 9.

Algorithm 1 *Terminal Reduction Algorithm*

```
1   ξ(M_ij) {
2       k = 0;
3       M_iter = M_ij;
4       while (1) {
            /* parallel on */
5           calculate T_r(M_iter); /* determine all terminal rows */
6           calculate T_c(M_iter); /* determine all terminal columns */
            /* parallel off */
7           if ((T_r(M_iter) == ∅) and (T_c(M_iter) == ∅)) break;
                /* if no more terminals */
            /* parallel on */
8           for each terminal row r_s ∈ T_r(M_iter),
                set all entries in row r_s to zero;
9           for each terminal column c_t ∈ T_c(M_iter),
                set all entries in column c_t to zero;
            /* parallel off */
10          k = k + 1;
11      }
12      M_{i,j+k} = M_iter;
13      return M_{i,j+k};
14  }
```

Algorithm 2 *Parallel Deadlock Detection Algorithm (PDDA)*

```
1   Deadlock_Detect_Matrix (γ_ij) {
2       M[s,t] = [α_st], where
3           s = 1, ... , m and t = 1, ... , n
4           α_st = r, if ∃ a request edge (p_t, q_s) ∈ E(γ_ij)
5           α_st = g, if ∃ a grant edge (q_s, p_t) ∈ E(γ_ij)
6           α_st = 0, otherwise.
7       M_{i,j+k} = ξ(M_ij); /* call Algorithm 1 */
8       if (M_{i,j+k} == [0]) { /* matrix of all zeros */
9           return 0; /* no deadlock */
10      } else {
11          return 1; /* deadlock detected */
12      }
13  }
```

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

We have proven that PDDA detects deadlock if and only if there exists a cycle in state γ_{ij} [29]. We have also proven that our hardware implementation of Algorithm 1 completes its computation in at most $2 \times min(m,n) - 3 = O(min(m,n))$ steps, where m is the number of resources and n is the number of processes [29].

4.2.2 Hardware implementation of PDDA: DDU

We here summarize the operation of PDDA in the hardware point of view, i.e., how to parallelize PDDA to implement in hardware (please see [29] for more information, which describes the sequence of DDU operations in great detail). As introduced in the previous subsection, a given system state γ_{ij} is equivalently represented by a system state matrix M_{ij} (shown in Equation 1) so that, based on M_{ij}, the DDU can perform the sequence of operations shown in Algorithms 1 and 2 and decide whether the given state has a deadlock or not.

$$\boldsymbol{M_{ij}} = \begin{bmatrix} \alpha_{11} & \dots & \alpha_{1t} & \dots & \alpha_{1n} \\ : & : & : & : & : \\ \alpha_{s1} & \dots & \alpha_{st} & \dots & \alpha_{sn} \\ : & : & : & : & : \\ \alpha_{m1} & \dots & \alpha_{mt} & \dots & \alpha_{mn} \end{bmatrix} = M_{iter} \quad (1)$$

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

Each matrix element α_{st} in M_{ij} represents one of the following: $g_{s \rightharpoonup t}$ (a grant edge), $r_{t \rightarrow s}$ (a request edge) or 0_{st} (no edge). Since α_{st} is ternary-valued, α_{st} can be minimally defined as a pair of two bits $\alpha_{st} = (\alpha^r_{st}, \alpha^g_{st})$. If an entry α_{st} is a grant edge g, bit α^r_{st} is set to 0, and α^g_{st} is set to 1; if an entry α_{st} is a request edge r, bit α^r_{st} is set to 1, and α^g_{st} is set to 0; otherwise, both bits α^r_{st} and α^g_{st} are set to 0. Hence, an entry α_{st} can be only one of the following binary encodings: 01 (a grant edge), 10 (a request edge) or 00 (no activity). Thus, M_{iter} in line 3 of Algorithm 1 can be written as shown in Equation 2.

$$\boldsymbol{M_{iter}} = \begin{bmatrix} (\alpha^r_{11}, \alpha^g_{11}) & \dots & (\alpha^r_{1t}, \alpha^g_{1t}) & \dots & (\alpha^r_{1n}, \alpha^g_{1n}) \\ : & : & : & : & : \\ (\alpha^r_{s1}, \alpha^g_{s1}) & \dots & (\alpha^r_{st}, \alpha^g_{st}) & \dots & (\alpha^r_{sn}, \alpha^g_{sn}) \\ : & : & : & : & : \\ (\alpha^r_{m1}, \alpha^g_{m1}) & \dots & (\alpha^r_{mt}, \alpha^g_{mt}) & \dots & (\alpha^r_{mn}, \alpha^g_{mn}) \end{bmatrix} \quad (2)$$

Finding terminal rows and terminal columns, which corresponds to lines 5 and 6 of Algorithm 1, requires three logical operations performed in sequence: (i) Bit-Wise-Or (BWO), (ii) eXclusive-OR (XOR), and (iii) OR. Two separate BWO operations, shown in Equation 3, take place through each row and each column of M_{iter}, all in parallel at the same time at each iteration in the DDU.

$$\begin{aligned} BWO^c_{iter} &= \forall t,\ (\alpha^r_{ct}, \alpha^g_{ct}) = \forall t,\ (\bigvee_{s=1}^{m} \alpha^r_{st},\ \bigvee_{s=1}^{m} \alpha^g_{st}) \\ BWO^r_{iter} &= \forall s,\ (\alpha^r_{rs}, \alpha^g_{rs}) = \forall s,\ (\bigvee_{t=1}^{n} \alpha^r_{st},\ \bigvee_{t=1}^{n} \alpha^g_{st}) \end{aligned} \quad (3)$$

where notation $\forall$ means **for all** and notation $\bigvee$ means Bit-Wise-Or of elements.

Then, from the results of two BWO operations, the XOR operations, shown in Equation 4, for each row and each column occur all in parallel.

$$\begin{aligned} XOR^c_{iter} &= \forall t,\ \tau_{ct} = \forall t,\ (\alpha^r_{ct} \oplus \alpha^g_{ct}) \\ XOR^r_{iter} &= \forall s,\ \tau_{rs} = \forall s,\ (\alpha^r_{rs} \oplus \alpha^g_{rs}) \end{aligned} \quad (4)$$

where $\oplus$ denotes eXclusive-OR.

Next, the OR operation, shown in Equation 5, produces a termination condition (i.e., the reducibility test of matrix M_{iter}, which corresponds to line 7 in Algorithm 1) at each iteration. That is, the termination condition represents whether a current matrix is further reducible or not. If T_{iter} equals '1,' meaning that more terminal edge(s) exist, the iterations continue. If the current matrix M_{iter} is irreducible (i.e., it has no terminal edges), T_{iter} will become '0'; thus, further iterations would accomplish nothing. This irreducibility condition can be written as

$$T_{iter} = (\tau_C \vee \tau_R) = (\bigvee_{t=1}^{n} \tau_{ct} \vee \bigvee_{s=1}^{m} \tau_{rs}). \quad (5)$$

Before finishing PDDA, one more important process remains: deadlock detection, which requires two more parallel logic operations. Equation 6 represents the existence of connect nodes in each column and in each row, respectively, involved in cycle(s).

$$\begin{aligned} AND^c_{iter} &= \forall t,\ \phi_{ct} = \forall t,\ (\alpha^r_{ct} \wedge \alpha^g_{ct}) \\ AND^r_{iter} &= \forall s,\ \phi_{rs} = \forall s,\ (\alpha^r_{rs} \wedge \alpha^g_{rs}) \end{aligned} \quad (6)$$

where $\wedge$ denotes Bit-Wise-And of elements.

Finally, Equation 7 produces the result of deadlock detection, which corresponds to lines 8-12 of Algorithm 2.

$$D_{iter} = (\phi_c \vee \phi_r) = (\bigvee_{t=1}^{n} \phi_{ct} \vee \bigvee_{s=1}^{m} \phi_{rs}) \text{ when } T_{iter} = 0 \quad (7)$$

4.2.3 Architecture of the Deadlock Detection Unit

The DDU consists of three parts as shown in Figure 13: matrix cells, weight cells and a decide cell. Part 1 is the system state matrix M_{ij} consisting of an array of matrix cells α_{st}. Part 2 consists of two weight vectors: (i) one column weight vector below the matrix cells and (ii) one row weight vector on the right side of matrix cells. The column weight vector is expressed as follows:

$$W^c = [\ w_{c1} \quad w_{c2} \quad \cdots \quad w_{ct} \quad \cdots \quad w_{cn}\ ] \quad (8)$$

where n is the number of processes, and $\forall t$, w_{ct} (each column weight cell) is a pair (τ_{ct}, ϕ_{ct}), representing whether the corresponding process node is a terminal node $(1, 0)$, a connect node $(0, 1)$, or neither $(0, 0)$. The row weight vector is expressed as follows:

$$W^r = [\ w_{r1} \quad w_{r2} \quad \cdots \quad w_{rs} \quad \cdots \quad w_{rm}\ ]^T \quad (9)$$

where m is the number of resources, and $\forall s$, w_{rs} (each row weight cell) is a pair (τ_{rs}, ϕ_{rs}), representing whether the corresponding resource node is a terminal node, a connect node, or neither. Part 3 is one decide cell D_{iter} at the bottom right corner of the DDU.

Figure 13 shows the architecture of the DDU for three processes and three resources. This DDU example has nine matrix cells (3×3) for all edge elements of M_{ij}, six weight cells (three for column processing and three for row processing), and one decide cell for making the decision of deadlock.

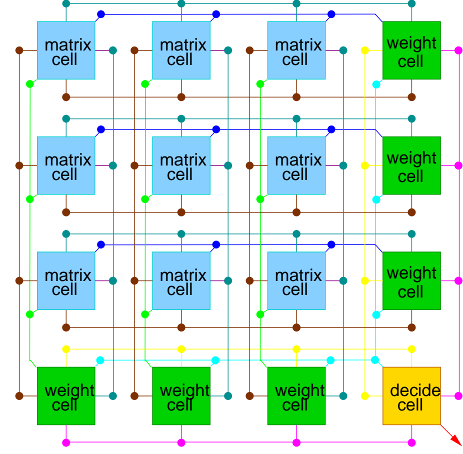


Figure 13 DDU architecture

4.2.4 Synthesis Result of the DDU

We used the Synopsys Design Compiler (DC) to synthesize the DDU with a $0.3\mu m$ standard cell library from AMIS [32]. Table 1 shows the synthesis results of five types of DDUs customized according to the number of processes and resources in an SoC. The fourth column, denoted "worst case # iterations," represents the number of worst case number of iterations for the corresponding DDU.

# processes × # resources	lines of Verilog	area in terms of two-input NAND gates	worst case # iterations
2×3	49	186	2
5×5	73	364	6
7×7	102	455	10
10×10	162	622	16
50×50	2682	14142	96

TABLE 1 SYNTHESIS RESULTS OF DDU

Please note that a system example using the DDU, including quantitative performance results, will be presented in Section 5.3.

4.3 New deadlock avoidance methodology

In our new approach to deadlock avoidance, we utilize the parallel deadlock detection algorithm (PDDA) and DDU. Unlike the DDU, 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.

The Deadlock Avoidance Unit (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.

4.3.1 New deadlock avoidance algorithm

Algorithm 3 shows our deadlock avoidance approach. We initially considered two other deadlock avoidance approaches but found Algorithm 3 to be better because it resolves livelock more actively and efficiently than two other approaches [28].

Let us proceed to describe Algorithm 3 step by step. When a process requests a resource from the DAU (line 2 of Algorithm 3), 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 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).

Algorithm 3 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              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         if any process is waiting for the released resource
18             if the grant of the resource would cause grant deadlock
19                 grant the resource to a lower priority process waiting
20             else
21                 grant the resource to the highest priority process waiting
22             end-if
23         else
24             make the resource become available
25         end-if
26 } end-case
}
```

When the DAU receives a resource release command from a process (line 16) and any process is waiting for the resource (line 17), 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 20), it becomes a fixed grant; thus the resource is granted to the highest priority requester (line 21). On the other hand, if the temporary grant causes G-dl (line 18), 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 19). 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.

4.3.2 Architecture of the DAU

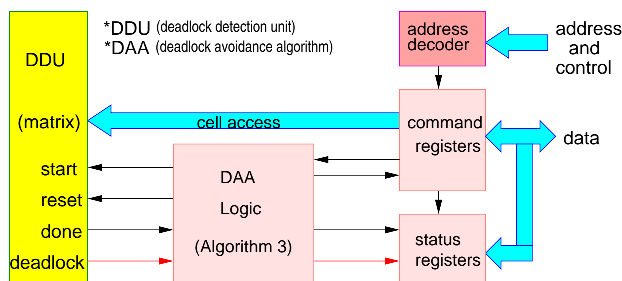


Figure 14 DAU architecture.

Figure 14 illustrates the DAU, implemented in Verilog HDL. The DAU consists of four parts: a Deadlock Detection Unit (DDU [27]), command registers, status registers and a unit implementing Algorithm 3 with a finite state machine. The command registers receive request and release 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.

4.3.3 Synthesis results of the DAU

We used the Synopsys Design Compiler (DC) [33] to synthesize the DAU for five processes and five resources with the QualCore Logic .25μm standard cell library [34]. The Synthesis result is shown in Table 2. 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 MPSoC contains

four PowerPC 755 PEs (1.7M gates each) and 16MB memory (33.5M gates), the area overhead in the MPSoC due to the DAU is about .005%.

Module Name	Lines of Verilog	Total Area	# Steps in Detection	# Steps in Avoidance
DDU 5x5	203	364	6	–
Others in Figure 14	344	1472	–	8
Total	547	1836(.005%)	–	$6 \times 5 + 8 = 38$
MPSoC		40.344M	–	

TABLE 2 SYNTHESIS RESULTS OF THE DAU.

4.4 Integrating DDU and DAU into the δ framework.

In Figure 3, for deadlock hardware components, after a user selects either the Deadlock Detection Unit (DDU) or the Deadlock Avoidance Unit (DAU), the GUI tool generates a deadlock IP component with a designated type and a specific size according to the number of tasks and resources specified in the Target Architecture window (see upper left of Figure 3).

5 Experimentation and results

In this section, we first explain the detailed base MPSoC for experimentation and various configured RTOS/MPSoCes. Then, we demonstrate performance comparisons among the RTOS/MPSoC systems with applications.

5.1 Base MPSoC for experimentation

Prior to inclusion of any hardware RTOS components, all configured RTOS/MPSoC for experimental simulations presented in this article have so called a base system consisting of four Motorola MPC755s and four resources implemented in Verilog HDL as introduced in Section 3.2.2 (note that PE cores are typically provided by simulation tool vendors such as processor support packages from Seamless CVE [35]). Each MPC755 has separate instruction and data L1 caches each of size 32KB. Four resources are a video interface (VI) device, a DSP, an IDCT unit and a wireless interface (WI) device. These four resources have timers, interrupt generators and input/output ports that are necessary to support our simulations. The base system also has a bus arbiter, a clock driver, a memory controller and a 16MB of shared memory. The master clock period of the bus system is 10 ns (100 MHz). Code for each MPC755 runs on an instruction-accurate (not cycle-accurate) MPC755 simulator provided by Seamless CVE [35].

The experimental simulations were carried out using Seamless Co-Verification Environment (CVE) [35] aided by Synopsys VCS [36] for Verilog HDL simulation and XRAY [37] for software debugging. We have used Atalanta RTOS version 0.3 [5], a shared-memory multiprocessor RTOS, introduced in Section 2.1.

5.2 Configured RTOS/MPSoCes for experimentation

Using the δ hardware/software RTOS design framework, we have configured various RTOS/MPSoC systems as shown in Table 3. All RTOS/MPSoC systems are generated primarily based on the base MPSoC described in the previous section.

System	Configured components on top of essential pure software RTOS
RTOS1	PDDA (i.e., Algorithms 1 and 2) in software (Section 4.2.1)
RTOS2	DDU in hardware (Sections 4.2.2 and 4.2.3)
RTOS3	DAA (i.e., Algorithm 3) in software (Section 4.3.1)
RTOS4	DAU in hardware (Section 4.3.2)
RTOS5	Pure RTOS with priority inheritance support (Section 2.1)
RTOS6	SoCLC with immediate priority ceiling protocol in hardware (Section 2.3.1)
RTOS7	SoCDMMU in hardware (Section 2.3.2)

TABLE 3 CONFIGURED RTOS/MPSOCES

5.3 Execution time comparison between RTOS1 and RTOS2

In this experiment, we wanted to identify the difference in an application executing using the DDU versus PDDA in software. In RTOS2, the MPSoC has a DDU for five processes and five resources. We devised an application example inspired by the Jini lookup service system [39], in which client applications can request services through intermediate layers (i.e., lookup, discovery and admission). Since the MPSoC, introduced in Section 5.1, has multiple processes and multiple resources, and Assumptions 1-3 in Section 3.2.3 can also easily be satisfied during the normal execution of the application, a deadlock is possible in such a system. Thus, this is an example of a practical application that can benefit from the DDU. In this experiment, we invoked one process on each PE and prioritized all processes, p_1 being the highest and p_4 being the lowest. The video frame we use for the experiment is a test frame whose size is 64 by 64 pixels. The IDCT processing time of the test frame takes approximately 23,600 clock cycles.

We show a sequence of requests and grants that finally leads to a deadlock as shown in Table 4 and Figure 15. Process p_1, running on PE1, requests both the IDCT and the VI at time t_1, which are then granted to p_1. After that, p_1 starts receiving a video stream through the VI and does IDCT processing. At time t_2, process p_3, running on PE3, needs and requests the IDCT and the WI to simultaneously convert a frame to an image and send the image through the WI. However, only the WI is granted to p_3 since the IDCT is unavailable. At time t_3, p_2 running on PE2 also requests the IDCT and WI hardware units, which are not available for p_2. When the IDCT is released by

p_1 at time t_4, the IDCT is granted to p_2 since p_2 has a higher priority than p_3. This last grant will lead to a deadlock in the SoC.

Time	Number	Events
t_0	e_0	The application starts.
t_1	e_1	p_1 requests IDCT and VI; IDCT and VI are granted to p_1 immediately.
t_2	e_2	p_3 requests IDCT and WI; WI is granted to p_3 immediately.
t_3	e_3	p_2 requests IDCT and WI. Both p_2 and p_3 wait IDCT.
t_4	e_4	IDCT is released by p_1.
t_5	e_5	IDCT is granted to p_2 since p_2 has a higher priority than p_3.

TABLE 4 A SEQUENCE OF REQUESTS AND GRANTS

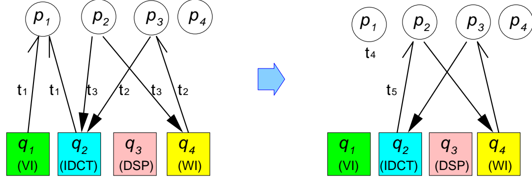


Figure 15 Events RAG

With the above scenario, we measured both the deadlock detection time Δ and the application execution time from the application start (t_0) until the detection of a deadlock in two cases: using (i) the DDU and (ii) PDDA in software. Note that the RTOS initialization time was excluded (i.e., the RTOS is assumed to be fully operational at time t_0). Table 5 shows that (i) in average the DDU achieved a 1408X speed-up over the software implementation of PDDA and that (ii) the DDU gave a 46% speed-up of application execution time over PDDA in software. The application invoked deadlock detection 10 times. Please note that the example application has not yet finished because of deadlock and that the algorithm run-time does not include the run-time of application programming interfaces. Note also that a different case where deadlock does not occur so early would of course not show a 46% speed-up, but instead would show a potentially far lower percentage speed-up; nonetheless, for critical situations where early deadlock detection is crucial, our approach can help significantly.

Method of Implementation	Algorithm Run Time*	Application Run Time*	Speedup[&]
DDU(hardware)	1.3	27714	$\frac{40523-27714}{27714} = 46\%$
PDDA in software	1830	40523	

*The time unit is a bus clock, and the values are averaged. [&]The speed-up is calculated according to the formula by Hennessy and Patterson [40].

TABLE 5 DEADLOCK DETECTION TIME AND APPLICATION EXECUTION TIME

5.4 Execution time comparison between RTOS3 and RTOS4

In this experiment, we wanted to identify the difference in an application executing using the DAU versus DAA in software. In RTOS4, the MPSoC has a DAU for five processes and five resources.

5.4.1 Application example I

We show a sequence of requests and grants that would lead to grant deadlock (G-dl) as shown in Figure 16 and Table 6. 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 IDCT, which are then granted to p_1. After that, p_1 starts receiving a video stream through VI and does IDCT processing. At time t_2, process p_3, running on PE3, requests IDCT 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 IDCT is unavailable. At time t_3, p_2 running on PE2 also requests IDCT and WI, which are not available for p_2. When IDCT is released by p_1 at time t_4, IDCT 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 IDCT to p_3 even though p_3 has a priority lower than p_2. Then, p_3 uses and releases IDCT and WI at time t_6. After that, IDCT and WI are granted to p_2 at time t_7, which finishes its job at time t_8.

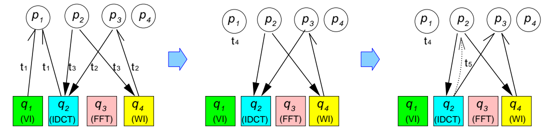


Figure 16 Events RAG (grant deadlock).

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.

5.4.2 Experimental result I

Table 7 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 (DAU or DAA) were invoked 12 times, respectively (since every request and release invokes one of the algorithms).

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 6 A SEQUENCE OF REQUESTS AND GRANTS THAT COULD LEAD TO GRANT DEADLOCK (G-DL).

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 time unit is a bus clock, and the values are averaged.

TABLE 7 EXECUTION TIME COMPARISON (G-DL).

5.4.3 Application example II

We show a sequence of requests and grants that would lead to request deadlock (R-dl) as shown in Figure 17. In this example, we assume the following. (i) Process p_1 requires resources q_1 (VI) and q_2 (IDCT) to complete its job. (ii) Process p_2 requires resources q_2 (IDCT) and q_3 (DSP). (iii) Process p_3 requires resources q_3 (DSP) and q_1 (VI). The detailed sequence is shown in Table 8. At time t_1, process p_1 requests and acquires q_1. At time t_2, process p_2 requests and acquires q_2. At time t_3, process p_3 requests and acquires q_3. After that, at time t_4, process p_2 requests q_3; since q_3 was already granted to p_3, and since the request does not cause R-dl, the request becomes pending. At time t_5, process p_3 requests q_1; since q_1 was already granted to p_1, and since the request does not cause R-dl, this request also becomes pending. 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}.

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.

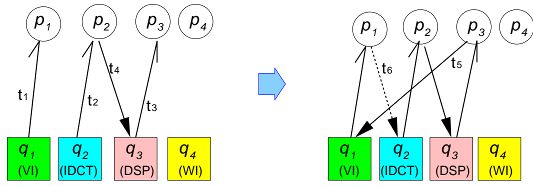


Figure 17 Events RAG (request deadlock).

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 8 A SEQUENCE OF REQUESTS AND GRANTS THAT WOULD LEAD TO REQUEST DEADLOCK (R-DL).

5.4.4 Experimental result II

Table 9 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 time unit is a bus clock, and the values are averaged.

TABLE 9 EXECUTION TIME COMPARISON (R-DL).

5.5 Execution time comparison between RTOS5 and RTOS6

This section presents the performance comparison between SoCLC (please see Section 2.3.1 for more detail) with Immediate Priority Ceiling Protocol (IPCP [17]) versus Atalanta

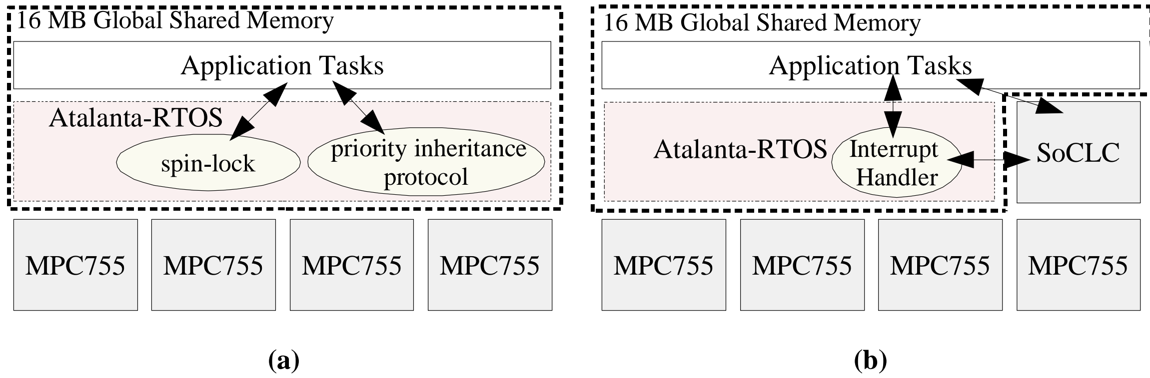


Figure 18 RTOS5 versus RTOS6.

RTOS with Priority Inheritance Protocol (AtalantaPI). Figure 18 depicts the two configured hardware/software architectures that we compare. The first RTOS/MPSoC architecture (Figure 18 (a)) comprises four MPC755 processors in hardware and the user-level application tasks plus RTOS5. The executable Atalanta RTOS includes the priority inheritance protocol and the spin-lock mechanism for lock-based synchronization of long critical sections (CSes) and short CSes. The second architecture (Figure 18 (b)), on the other hand, comprises four MPC755 processors, RTOS6 (i.e., the SoCLC in hardware plus the Atalanta RTOS in software) and the user-level application tasks. That is, the Atalanta RTOS of the second architecture utilizes the priority inheritance protocol (which is part of the lock-based long CS synchronization) and the lock-based short CS synchronization facility that are implemented as part of the SoCLC in hardware. The tasks in this comparison represent a robot control application and an MPEG decoder. Figure 19 illustrates the algorithmic model of the robot control application.

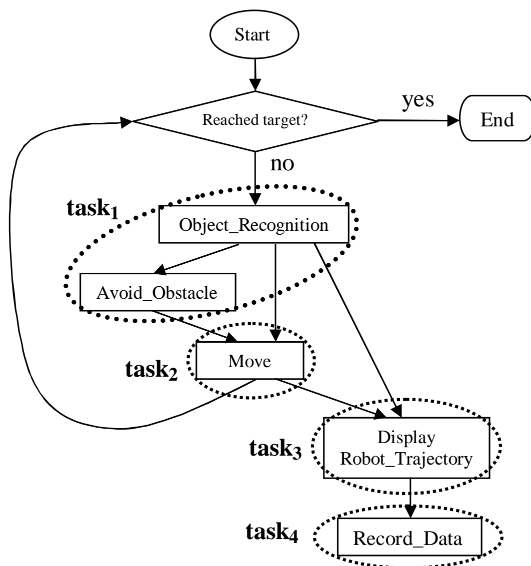


Figure 19 Robot example.

The first task detects the obstacles over the path via a sensor operation and then computes the coordinates of the next path to be taken by the robot to avoid a collision with the obstacle. As seen from Figure 19, *Object Recognition* and *Avoid Obstacle* parts of the model have been assigned to $task_1$, which is the highest priority task with critical hard real-time requirements. The worst case response time (WCRT) of $task_1$ is 250μs; missing the deadline of $task_1$ causes instability in the sensor function and tracking to fail. Also seen in Figure 19, $task_2$ handles the movement of the robot according to the position information already determined by $task_1$. $Task_2$ is the second highest priority task with firm real-time requirements and has a response time of 300μs. Missing the deadline of $task_2$ causes the speed of the robot to decrease and/or gouging or breakage. $Task_3$ and $task_4$, on the other hand, have relatively soft timing requirements and are responsible for the robot trajectory display and recording. The WCRT of $task_3$ and $task_4$ are 300μs and 600μs, respectively. Finally, the MPEG decoder task, $task_5$, is the lowest priority task in the system and has a soft timing requirement.

In the simulations, these five tasks execute as follows: $task_1$ runs on PE1 and it has a priority of 1 (highest priority task), $task_2$ is the second priority task with priority 2 and it runs on PE2, $task_3$ also runs on PE2 with priority 3, $task_4$ runs on PE3 and $task_5$ runs on PE4. Figure 20 shows the execution traces of $task_1$, $task_2$ and $task_3$. As seen in Figure 20, during the time that $task_1$ is waiting for $task_3$ to release the lock, $task_1$ (highest priority task) is prevented from having unbounded blocking. Because, with IPCP, $task_3$'s priority is raised to the ceiling priority immediately after acquiring the lock. Therefore, when $task_2$ (whose priority is higher than $task_3$) arrives, $task_2$ cannot preempt $task_3$, so $task_3$ runs on PE2 until $task_3$ completes the CS and releases the lock.

For performance comparison, the lock latency, lock delay and overall execution times for both architectures shown in Figure 18 were measured. The first architecture does not include SoCLC, and is named as the "RTOS5" case; the second architecture includes SoCLC, and is named as the "RTOS6" case. As seen from Table 10, RTOS6 (the SoCLC with the priority inheritance implemented in hardware) achieves a 79%

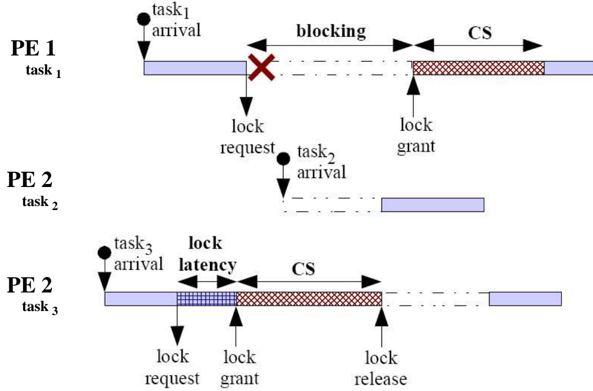


Figure 20 $Task_3$ inherits $task_1$'s priority during the time that $task_3$ executes its CS. After completing CS, $task_3$ yields the PE2 to $task_2$.

speed-up (i.e., 1.79X) in the lock latency, a 75% speed-up (i.e., 1.75X) in the lock delay and a 43% speed-up (i.e., 1.43X) in the overall execution time when compared to RTOS5 (Atalanta RTOS with the priority inheritance software implementation).

(time in clock cycles)	RTOS5	RTOS6	Speedup
Lock Latency	570	318	1.79 X
Lock Delay	6701	3834	1.75 X
Overall Execution	112170	78226	1.43 X

TABLE 10 SIMULATION RESULTS OF THE ROBOT APPLICATION.

Please note that we assume three cycles of the system bus clock (including bus arbitration) are needed to access the first word in the 16 MB global memory (if the transaction is a burst transaction, the successive words of the burst are accessed each in one clock cycle).

5.6 Execution time comparison between RTOS5 and RTOS7

This section demonstrates the performance comparison between RTOS5 and RTOS7. For the performance comparison, several benchmarks taken from the SPLASH-2 application suite have been used: Blocked LU Decomposition (LU), Complex 1D FFT (FFT) and Integer Radix Sort (RADIX) [41, 42].

The selected benchmarks source files were modified to replace all the static memory arrays by arrays that are dynamically allocated at run time and deallocated upon completion. In this way, the benchmarks could be dynamically downloaded and run on a handheld device, which is the kind of ability the SoCDMMU research focuses on.

Table 11 shows the execution time of the benchmarks in clock cycles and the total number of cycles consumed in memory management when the benchmarks use a conventional memory allocation/deallocation techniques (*glibc* [43, 44] *malloc()* and *free()*).

Table 12 shows the same information introduced in Table 11 but with the benchmarks using the SoCDMMU for memory allocation/deallocation. Also, Table 12 shows the reduction in the memory management execution time due to using the SoCDMMU instead of using *glibc malloc()* and *free()* functions. This reduction in the memory management execution time yields speed-ups in the benchmark execution time. As we can see in Table 12, using the SoCDMMU tends to speed up the application execution time and this speed-up is almost equal to the percentage of time consumed by conventional software memory management techniques.

Benchmark	Total exe. time (cycles)	Memory management time (cycles)	% of time used for memory management
LU	318307	31512	9.90%
FFT	375988	101998	27.13%
RADIX	694333	141491	20.38%

TABLE 11 EXECUTION TIME OF SOME SPLASH-2 BENCHMARKS USING *glibc malloc()* AND *free()*.

Bench-mark	Total Time (cycles)	Memory mgmt. time (cycles)	% of time used for memory mgmt.	% Reduction in time used to manage memory	% Reduction in benchmark exe. time
LU	288271	1476	0.51%	95.31%	9.44%
FFT	276941	2951	1.07%	97.10%	26.34%
RADIX	558347	5505	0.99%	96.10%	19.59%

TABLE 12 EXECUTION TIME OF SOME SPLASH-2 BENCHMARKS USING THE SOCDMMU.

6 Conclusion

This article presents a methodology of hardware/software partitioning of operating systems with the δ hardware/software RTOS/MPSoC codesign framework that has been used to configure and generate simulatable RTOS/MPSoC designs having both appropriate hardware and software interfaces as well as system architecture. The δ framework is specifically designed to help RTOS/MPSoC designers very easily and quickly explore their design space with available hardware and software modules so that they can efficiently search and discover a couple of optimal solutions matched to the specifications and requirements of their design before an actual implementation.

We have configured, generated and simulated various RTOS/MPSoC systems with available hardware/software RTOS components such as System-on-a-Chip Lock Cache (SoCLC), SoC Dynamic Memory Management Unit (SoCDMMU), Deadlock Detection Unit (DDU), Deadlock Avoidance Unit (DAU), and equivalent software modules, respectively. From the simulations using Seamless CVE from Mentor Graphics, we show that our methodology is a viable approach to rapid hardware/software partitioning of operating systems. In addition, we demonstrated the following with experiments. (i) An RTOS/MPSoC system with the DDU

achieved about a 1400X speed-up of the deadlock detection time and a 46% speed-up of an application execution time over an RTOS/MPSoC system with a deadlock detection method in software. (ii) A system with the DAU reduced the deadlock avoidance time by 99% *(about 300X)* and an application execution time by 44% as compared to a system with a deadlock avoidance algorithm in software. (iii) A system with the SoCLC shows about a 75% speed-up in the lock handling and a 43% speed-up in the overall execution time when compared to a system implemented priority inheritance and lock handling in software. (iv) A system with the SoCDMMU reduced about 20% of time used for memory management and execution times of benchmarks 9.44% or more over a system without the SoCDMMU.

Acknowledgements

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.

References

[1] J. Lee, K. Ryu and V. Mooney, "A framework for automatic generation of configuration files for a custom RTOS," *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms* (ERSA'02), pp. 31-37, June 2002.

[2] J. Lee, V. Mooney, A. Daleby, K. Ingstrom, T. Klevin and L. Lindh, "A comparison of the RTU hardware RTOS with a hardware/software RTOS," *Proceedings of the Asia and South Pacific Design Automation Conference* (ASPDAC 2003), pp.683-688, Jan. 2003.

[3] V. Mooney and D. Blough, "A hardware-software real-time operating system framework for SOCs," *IEEE Design and Test of Computers*, pp. 44-51, Nov.-Dec. 2002.

[4] V. Mooney, "Hardware/software partitioning of operating systems," in the book *Embedded Software for SoC*, edited by A. Jerraya, S. Yoo, D. Verkest and N. Wehn, published by Kluwer Academic Publishers, Boston, MA, USA, pp. 187-206, Sep. 2003.

[5] 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.

[6] 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, Fall 2004.

[7] K. Ryu and V. Mooney, "Automated bus generation for multiprocessor SoC Design," *Proceedings of the Design Automation and Test in Europe Conference* (DATE'03), pp. 282-287, March 2003.

[8] K. Ryu and V. Mooney, "Automated bus generation for multiprocessor SoC design," to be published in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(11), pp. 1531-1549, Nov. 2004.

[9] K. Ryu, "Automatic generation of bus systems," Ph.D. Thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, USA, Summer 2004.

[10] B. Akgul and V. Mooney, "PARLAK: Parametrized Lock Cache generator," *Proceedings of the Design Automation and Test in Europe Conference* (DATE'03), pp. 1138-1139, March 2003.

[11] B. Akgul, "The System-on-a-Chip Lock Cache," Ph.D. Thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, USA, Spring 2004.

[12] M. Shalan, E. Shin and V. Mooney, "DX-Gt: Memory management and crossbar switch generator for multiprocessor system-on-a-chip," *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies* (SASIMI'03), pp. 357-364, April 2003.

[13] M. Shalan, "Dynamic memory management for embedded real-time multiprocessor system-on-a-chip," Ph.D. Thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, USA, Fall 2003.

[14] B. Akgul, J. Lee and V. Mooney, "A System-on-a-Chip Lock Cache with task preemption support ," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (CASES'01), pp. 149-157, Nov. 2001.

[15] B. Akgul and V. Mooney, "The System-on-a-Chip Lock Cache," *Trans. on Design Automation for Embedded Systems*, 7(1-2), pp. 139-174, Sep. 2002.

[16] B. Akgul, V. Mooney, H. Thane and P. Kuacharoen, "Hardware support for priority inheritance," *Proceedings of the IEEE Real-Time Systems Symposium* (RTSS'03), pp.246-254, Dec. 2003.

[17] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. on Computers*, 39(9), pp. 1175–1185, Sep. 1990.

[18] M. Shalan and V. Mooney, "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," *Proceedings of the Tenth International Symposium on Hardware/Software Codesign* (CODES'02), pp. 79-84, May 2002.

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

[20] A. Shoshani and E. Coffman Jr., "Detection, Prevention and recovery from deadlocks in multiprocess, multiple resource systems," *4th Annual Princeton Conf. on Information Sciences and System*, March 1970.

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

[22] T. Leibfried Jr., "A deadlock detection and recovery algorithm using the formalism of a directed graph matrix," *Operation Systems Review*, 23(2), pp. 45-55, April 1989.

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

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

[25] E. Coffman, M. Elphick and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, pp. 67–78, June 1971.

[26] F. Belik, "An efficient deadlock avoidance technique," *IEEE Trans. on Computers*, 39(7), pp. 882–888, July 1990.

[27] P. Shiu, Y. Tan and V. Mooney, "A novel parallel deadlock detection algorithm and architecture," *Proceedings of the 9th International Symposium on Hardware/Software Codesign* (CODES'01), pp. 30-36, April 2001.

[28] J. Lee and V. Mooney, "A novel deadlock avoidance algorithm and its hardware implementation," *Proceedings of the 12th International Conference on Hardware/Software Codesign and System Synthesis* (CODES/ISSS'04), pp.200–205, Sep. 2004.

[29] 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.

[30] 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.

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

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

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

[34] QualCore Logic. http://www.qualcorelogic.com/.

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

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

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

[38] ModelSim HDL Simulator, http://www.model.com/.

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

[40] J. Hennessy and D. Patterson, *Computer architecture - a quantitative approach*. Morgan Kaufmann Publisher, Inc., San Francisco, CA, 1996.

[41] Stanford University, Stanford Parallel Applications for Shared Memory (SPLASH). http://www-flash.stanford.edu/apps/SPLASH/.

[42] S. Woo, M. Ohara, E. Torrie, J. Singh and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," Proceedings of the 22nd International Symposium on Computer Architecture, pp. 24–36, June 1995.

[43] The Free Software Foundation, The GNU project, the GCC compiler. http://gcc.gnu.org/.

[44] The Free Software Foundation, The GNU project, the GNU C library. http://www.gnu.org/software/libc/manual/.