

# Reducing Occurrences of Priority Inversion in MSoC's Using Dynamic Processor Priority Assignment

Mikael Collin, Mladen Nikitovic, and Christer Norström  
{mci,mnc,cen}@mdh.se

Department of Computer Engineering, Mälardalen University, Sweden

## Abstract

*In a singleprocessor real-time system the question whether a taskset is schedulable or not is essential and not always easy to answer, and it is even more challenging in multiprocessor real-time systems. To determine the schedulability of a taskset the maximum blocking time for each task must be known. When considering schedulability of tasks on a multiprocessor with a shared bus, both local blocking and interprocessor blocking must be known and considered. To reduce the time high priority tasks are blocked by low priority task on a shared bus multiprocessor system we propose a dynamic priority based arbitration of processors. In our proposal, a processor's priority is a direct mapping of the relative priority of all tasks currently executing. The mapping of priorities onto processors is managed by a hardware operating system kernel which continuously updates the arbiter with priority changes. The information shared between the hardware operating system and the arbiter is done in parallel and does not put additional strain on bus traffic. The solution is implemented by augmenting an hardware real-time scheduler and an arbiter unit with added functionality. Systems-on-a-Chip (SoC's) offers the design freedom needed for our proposed solution to reduce occurrences of priority inversion in Multiprocessor SoC (MSoC).*

## 1 Introduction

The blocking time of a task is traditionally the time a task is waiting for a shared resource to be available when some other task is currently using it. This type of blocking is referred to as *local blocking*. In multiprocessor systems, tasks can block each other when requesting access to globally shared resources. This occurrence, referred to as *interprocessor blocking*[3], is the time a task executing on one processor must wait for a shared resource to be freed by a task executing on another processor. Interprocessor blocking in a shared bus multiprocessor consists of two types of blocking: *remote blocking* [6, 2] occurs when a task has to wait for tasks located on another processor executing a critical section using a globally shared resource, and

*implicit blocking* occurs when two or more processors requests bus mastership simultaneously or when the bus is already owned by another processor. *Priority inversion* is an additional property to both local and interprocessor blocking and occurs when a high priority task is blocked by a low priority task. This occurrence should be avoided to minimize the response time of high priority tasks.

The number of papers addressing interprocessor blocking are sparse. Much of the articles regarding multiprocessor real-time systems and dynamic arbitration has focused on dynamic scheduling of tasks or assigning dynamic priority to processors according to predefined algorithms. Little effort has been done on investigating the possibility to let the task scheduling mechanism control the processor's priority during runtime.

Chang, et al[3] proposes a scheduling scheme to minimize interprocess blocking in systems utilizing static priority arbitration, where high priority tasks are located on processors with high arbitration priority. The scheduling of tasks must take the arbitration policy into account. This kind of static allocation of tasks to processors makes process migration impractical.

Zhang et al[8] presents a solution with dynamic processor priorities in a system utilizing daisy-chain arbitration. By enhancing the daisy-chain arbitration with a token ring functionality they were able to implement several priority arbitration algorithms. Decision of which algorithm to use is based on application characteristics.

Futurebus+ IEEE std896.1 and its support for real-time applications was analyzed by Sha, et al[5]. Their analysis states that Futurebus+ supports prioritized arbitration with 256 different priorities which can be dynamically modified by software in order to map executing tasks priority onto the arbitrated units. However, 256 levels of priorities are excessive since most busses only supports 8 levels.

Enhanced PCI (EPCI), is proposed by Scottis et al[4] where the software operating system schedules and assigns priorities to communication devices and primitives. A PCI bus is augmented with a priority arbitrated *stream bus*, where the software OS can alter priorities of messages via a programmable arbiter.

To reduce priority inversion scenarios we propose a solution that makes use of the total system state available from a hardware real-time kernel’s scheduler unit. The system state information is used to achieve a one-to-one mapping of all currently executing task’s relative priorities onto processor priority. Our solution provides system designers the means to use operating system information to dynamically control bus arbitration. Our solution targets both homogeneous and heterogeneous shared bus multiprocessors, and is applicable for both centralized and decentralized arbiters. Section 2 describes the hardware architecture. In section 3 the proposed solution and architectural modifications are presented and described. Section 4, summarizes the paper and present further development.

## 2 Architecture

The proposed solution is based on communication between a hardware real-time operating system and an arbiter unit via an added *priority bus*(Figure 1). The system designer has to have enough freedom to implement this kind of non-standard construction and MSOC’s are therefore especially suitable target for our proposed solution.

### 2.1 Hardware Real-Time Kernel

The proposal of a dynamic priority arbiter relies on a hardware real-time operating system to update information to the arbiter unit during run-time. One such operating system, namely the *Real-Time Unit* (RTU) has been developed at Mälardalen University[7, 1]. It is a centralized multitasking real-time kernel capable of managing a generic number of tasks and priority levels, which today can be mapped onto a maximum of three processors. It can schedule the tasks either statically or dynamically. Communication between RTU and processor is accomplished by register reads and writes. The RTU forces processors to make a taskswitch by issuing an interrupt. All the information needed to schedule the entire taskset in the system is located within the RTU which enables the arbiter unit to use this information during the arbitration cycle. The RTU

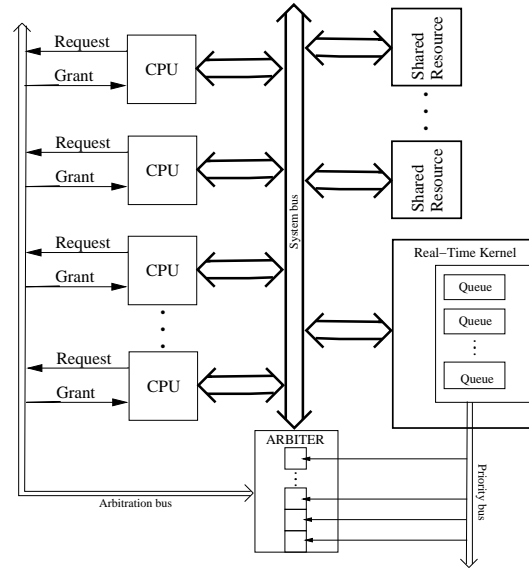


Figure 1: A view of a multiprocessor system with the proposed solution.

also supports load balancing by migrating tasks among processors.

### 2.2 Dynamic Priority Arbitration Unit

The arbitration unit implements a *priority based algorithm*, which uses the information from the RTU to determine which processor has the highest priority. Priority changes of processors are considered by the arbiter during run-time, making it *dynamic*. The values coming from the priority bus are written to local registers that represent a processor’s priority. During each arbitration cycle, the arbiter uses its arbitration algorithm, taking the current processor priorities into account.

## 3 Proposed Solution

The proposed solution is based on sharing of priority information between the RTU and the arbitration unit via an unidirectional priority bus which enables the RTU to send processor priorities to the arbiter during run-time. The RTU has information about which task currently is executing on each processor. This information enables the RTU to assign processor priorities during runtime whenever an executing task is replaced on a processor.

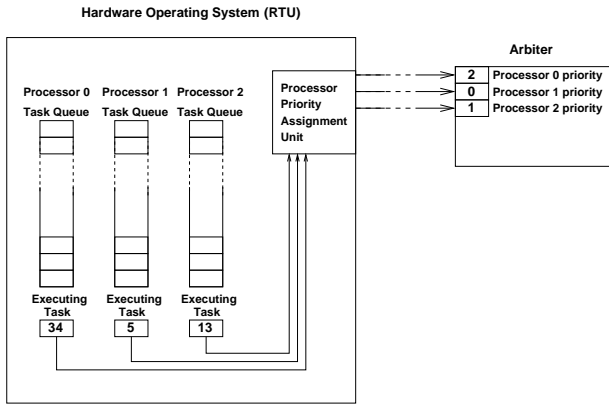


Figure 2: Internal modifications made to the RTU and arbiter.

### 3.1 Hardware Modifications

The current version of the RTU need to be slightly modified to be able to assign processor priorities based on currently executing tasks. Today, the interior of the RTU contains task queues for each processor. An extra unit within the RTU is needed that examines the priorities of the currently executing tasks and maps their relative priorities to their processors. Also, the arbiter unit need registers that represents each processor in the system. Whenever the RTU sends priority information via the priority bus, the values are written to the arbiter's processor registers, making the new processor priorities available in the next arbitration cycle. Figure 2 shows an example where the RTU schedules tasks on a multiprocessor system with three processors. Each processor has its own task queue within the RTU. One task from each queue is scheduled to execute on its processor. In this example, each tasks priority is shown, where the highest number equals to the highest priority. The task's priority values are read by the added function, *The Processor Priority Assignment Unit*, which calculates the processor priorities with respect to their executing tasks priority. In this case, processor 0 receives the highest priority value (2), processor 1 receives the lowest priority value (0), and processor 2 receives the intermediate priority.

### 3.2 Avoiding Priority Inversion

This hardware solution eliminates the priority inversion scenarios that occurs due to statically or poorly assigned dynamic processor priorities. Our proposed solution makes it possible to update processors priorities every scheduling cycle, which has microsecond granularity, avoiding scenarios described in Figure 3, where a

system with static processor priority is compared to our proposed solution. A system with statically prioritized processors is shown in figure 3a) where a high priority task can be located on a processor with low priority and vice versa. When both processors try to request the bus at the same time, the processor with the higher priority is granted ownership, which lets the task with low priority perform operations on the bus instead of the task with the high priority. In figure 3b), using our proposed solution, the processors priority is always reflected by the currently executing tasks priority and therefore the priority inversion scenario previously described will be avoided. Figure 4 shows a multiproces-

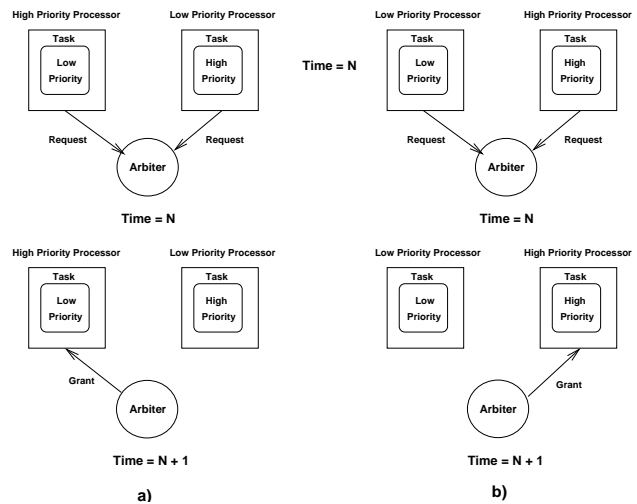


Figure 3: An example of how priority inversion is avoided by assigning processor priorities dynamically.

processor system with three processors  $\{P_1, P_2, P_3\}$  executing three tasks  $\{T_1, T_2, T_3\}$ , a RTU, and a shared resource.  $T_1$  accesses the globally shared resource.  $T_2$  writes to a memory located on the third processor.  $T_3$  performs no global actions and executes only locally. Global actions such as accessing a shared resource or writing to a non-local memory is defined as a *transaction* throughout this example. Transactions can either be *atomic* or *non-atomic*. Atomic transactions locks the bus during the entire transaction and does not release the bus until every instruction in the transaction has finished. In non-atomic transactions every single assembler instruction that writes or reads values on the bus must perform arbitration in order to become bus master, thus increasing the granularity of the arbitration cycle which can increase total bus and processor utilization.  $T_1$ 's transaction consists of 2 global accesses(i.e load instructions), followed by 2 local computations, and 3 global accesses(i.e store instructions). The transaction

of T2 consists of 4 global accesses only. Figure 5 shows an execution trace of  $\{T_1, T_2\}$  regarding bus ownership, and how priority inversion scenarios affect the response time of high priority tasks in the system. Tasks and processors can have either high (H) or low (L) priority. Column 1 to 4 shows the execution of  $\{T_1, T_2\}$  in a system with static processor priorities both with atomic (A) and non-atomic transactions (N), whereas column 5 to 7 shows possible execution scenarios in a system with dynamic processor priorities. Shaded columns indicates that there was priority inversion in the system, yielding an increase in the response time of high priority task. The conclusion is that the response time of high priority tasks is reduced because the occurrences of priority inversion in static processor priority systems can be avoided in our proposed solution.

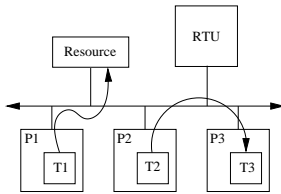


Figure 4: An example of a multiprocessor system.

Static processor priority				Dynamic processor priority		
$T_1=L$	$T_1=H$	$T_1=L$		$T_1=H$	$T_1=L$	
$T_2=H$	$T_2=L$	$T_2=H$		$T_2=L$	$T_2=H$	
$P_1=L$	$P_1=L$	$P_1=H$		$P_1=H$	$P_1=L$	
$P_2=H$	$P_2=H$	$P_2=L$		$P_2=L$	$P_2=H$	
A/N	A/N	N	A	N	A	A/N
$T_2$	$T_2$	$T_1$	$T_1$	$T_1$	$T_1$	$T_2$
$T_2$	$T_2$	$T_1$	$T_1$	$T_1$	$T_1$	$T_2$
$T_2$	$T_2$	$T_2$	$T_2$	$T_2$	$T_2$	$T_2$
$T_2$	$T_2$	$T_2$	$T_2$	$T_2$	$T_2$	$T_2$
$T_1$	$T_1$	$T_1$	$T_1$	$T_1$	$T_1$	$T_1$
$T_1$	$T_1$	$T_1$	$T_1$	$T_1$	$T_1$	$T_1$
-	-	$T_2$	$T_2$	$T_2$	$T_2$	-
-	-	$T_2$	$T_2$	$T_2$	$T_2$	-
$T_1$	$T_1$	$T_2$	$T_2$	$T_2$	$T_2$	$T_1$
$T_1$	$T_1$	$T_2$	$T_2$	$T_2$	$T_2$	$T_1$
$T_1$	$T_1$	$T_2$	$T_2$	$T_2$	$T_2$	$T_1$
①	②	③	④	⑤	⑥	⑦

Figure 5: An execution trace of tasks in static and dynamic processor priority systems.

## 4 Summary and Future Work

Although no results have been presented yet, we expect our proposed hardware solution to reduce the number of priority inversion occurrences in a multiprocessor

system. This is accomplished by letting the hardware operating system continuously provide the arbitration unit with processor priorities. Also, this solution enables the system to take full benefit of task migration due to the one-to-one mapping between task and processor priorities. A formalization of a tasks response time in a multiprocessor system is needed that takes bus transaction time, local and interprocessor blocking into account before we can prove the effectiveness of our solution. There are priority inversion scenarios that our solution cannot handle today, and those occur when a low priority task already has bus mastership and a task with higher priority must wait until the bus is available again. This scenario could be handled if it was possible to preempt the current bus transaction conducted by the low priority task. Such hardware construction can be implemented together with our solution to further reduce possible priority inversion scenarios.

## References

- [1] J. Adomat, J. Furunas, L. Lindh, and J. Starner. Real-time kernel in hardware rtu: A step towards deterministic and high-performance real-time systems. page 509. Proceedings of 22nd EUROMICRO Conference., 1996.
- [2] C. Y. Choi and H. Shin. Impact of bus arbitration on the schedulability of real-time shared-bus multiprocessors. page 602. TENCON'94. 9th IEEE Region 10's Conference, 1994.
- [3] C. Y. Choi, H. Shin, and Y. Cho. Interprocessor-blocking independent static task allocation for shared-bus real-time multiprocessors. page 53. Proceedings of 6th Euromicro Workshop on Real-Time Systems, 1994.
- [4] M. G. Scottis, M. Krunz, and M. M.-K. Liu. Enhancing the pci bus to support real-time streams. page 303. IEEE Performance, Computing and Communications Conference, 1999.
- [5] L. Sha, R. Rajkumar, and J. Lehoczky. Real-time scheduling support in futurebus+. page 331. Proceedings of 11th Real-Time Systems Symposium, 1990.
- [6] L. C. Shu and M. Young. Chopping and versioning real-time transactions to avoid remote blocking. page 93. Proceedings of 7th Real-Time Computing Systems and Applications Conference, 2000.
- [7] J. Starner, J. Adomat, J. Furunas, and L. Lindh. Real-time scheduling co-processor in hardware for single and multiprocessor systems. page 164. Proceedings of 8th Euromicro Workshop on Real-Time Systems, 1996.
- [8] C. N. Zhang, W. Chan, and M. Bachtar. Token ring arbitration circuits for dynamic priority algorithms. page 74. Proceedings of 37th Midwest Symposium on Circuits and Systems, 1994.