

Timing Implications of Sharing Resources in Multicore Real-Time Automotive Systems

2010-01-0454

Published
04/12/2010Mircea Negrean, Simon Schliecker and Rolf Ernst
Technische Univ. Braunschweig

Copyright © 2010 SAE International

ABSTRACT

The topic of timing has already been recognized as a major challenge when designing safety-critical automotive architectures. Consequently the availability of appropriate performance and timing analysis methods is key to building reliable automotive electric and electronics (E/E) and software architectures. Due to the potential performance increase, power reduction and cost-efficiency multicore solutions for automotive real-time environments receive growing attention. But the prediction of the timing behavior for multicore electronic control unit (ECU) systems becomes more complicated. Even in setups with static task-to-processor mapping, the execution of the tasks is usually not independent. The use of the same physical hardware, such as memories, coprocessors, or network components, makes inter-core interference unavoidable and may introduce hard-to-find timing problems including missed deadlines that can finally make the entire system fail. With this work we anticipate new issues on the timing prediction in the upcoming automotive multicore ECUs, we discuss difficulties and solutions in the evolution from distributed single-core systems to multicore systems and also provide a modeling and analysis approach of complex real-time multicore systems with shared resources.

1. INTRODUCTION

Timing and performance are key aspects of the system architecture design which have only recently received broad attention in the automotive industry. Intensive work of the AUTOSAR (Automotive Open System Architecture) timing group [5] and other industry-driven research projects (TIMMO, ALL-TIMES) with the scope to develop a formal language and a methodology for timing and performance design in the automotive domain, indicate the industry's awareness to these issues.

Current automotive systems are very heterogeneous being composed of several ECUs, several CAN or/and FlexRay bus segments connected via one (or multiple) gateway(s). Focusing on the computational units, most vehicle ECUs contain a single-core CPU that is possibly enhanced with specialized peripheral processors to offload some of the time critical and control functions. But the number and complexity of electronic functions in today's cars is ever increasing. Driven by this increase in functions and by the power limitations of single-core processor systems, the semiconductor industry starts offering multicore solutions for automotive processors today or in the near future [11, 21, 14]. The new multicore electronic control units (ECUs) are aimed to provide new functionality by clustering previously distributed applications into a single chip or to allow the parallelization of complex computations over multiple cores, for example in relatively high-performance domains such as engine control or advanced driver assistance systems. In this case, the multicore architecture will be used to distribute the load of a single application over several cores.

The shared memory communication of a task system that was previously implemented on a single-core is in this case distributed over several cores, which implies the need to share function units, such as coprocessors. Multicore adds another level of topological complexity to the automotive domain that is known from the system-on-chip (SOC) world for quite some time: shared resources (e.g. shared memories), crossbar switches and on-chip communication. Traditionally, safe sharing and communication in automotive task systems uses locking and synchronization mechanisms that are supported by the operating system. When mapping such task systems to multicore architectures the prediction of the timing behavior becomes more complicated. As sharing resources among cores makes inter-core interference unavoidable new types of arbitration conflicts emerge that require new scheduling and analysis algorithms. Formal performance

analysis methods which have been successfully applied for single-core distributed systems need to be extended in order to allow the timing analysis of the foreseeable multicore architectures.

Relying on current automotive practice and on related work of the real-time research community this paper discusses the challenges which arise when moving towards multicore architectures, highlights the impact of the shared resource usage on the system's timing and proposes a modeling and analysis approach for multicore ECUs with shared resources. The method covers the current and foreseeable automotive practice regarding standards (OSEK, AUTOSAR [6]), priority assignments (static and often manually assigned), and inter-task synchronization (through lock-based mechanisms). Dedicated experiments will underline the timing implication of multicore components and the applicability of the developed approach.

1.1. CURRENT SITUATION AND FORESEEABLE EVOLUTION TOWARDS MULTICORE

A large part of today's time critical ECU software has been developed for single processor systems. Many applications are arbitrated with real-time capable operating systems based on the OSEK/VDX specification [19], which defines priority based scheduling and allows resource synchronization via locks administered according to the Priority Ceiling Protocol (PCP). This lock based approach has been inherited by the more recent AUTOSAR OS specifications [3, 4]. Automotive suppliers and manufacturers are highly interested in re-using previous software generations and configurations, not only for economic reasons, but also because property changes are not a trivial task in a value chain that often involves many different departments and companies. In the past, this has often implied that non-optimal design choices are accepted for the sake of compatibility, for example by respecting legacy priority assignments and mapping decisions. In this environment, we expect that partitioned priority-driven multiprocessor scheduling will be used in the first evolutionary approach towards multicore architectures in order to maintain backwards compatibility. For the same reason, the availability of a lock-based shared resource arbitration policy is a requirement to avoid significant changes to existing designs and the current design practice.

1.2. REAL-TIME ANALYSIS METHODS

Several policies for multiprocessor scheduling have been proposed by the research real-time community. These can be classified into two different classes, depending on the flexibility of scheduling any given task: the *partitioned* and the *global / non-partitioned* approach.

In the *partitioned* scheduling approach tasks are statically assigned to the processors, which are independently scheduled at run-time. The mapping can be done manually (a common approach in automotive projects) or according to a bin-packing heuristic [17]. Schedulability tests are then used on each processor to verify whether tasks meet their deadlines. In the *global* scheduling approach, the scheduler maintains a single scheduling queue of ready tasks, from which tasks are dynamically dispatched on the available processors and possibly migrated during execution. As opposed to the partitioned approach different instances of the same task, or even different parts of the same instance, can execute on different processors. Various new scheduling policies have been proposed that take advantage of the additional freedom, but practical experiments taking real operating systems and architectures with caches into account, show numerous side effects leading to variable results [10] that are hard to predict.

In order to determine the timing performance of multiprocessor systems for both mentioned scheduling approaches, different feasibility and schedulability tests were proposed considering different scheduling policies such as static priority preemptive (SPP) [1] or global earliest deadline first (G-EDF) [7] and different task models, like periodic or sporadic [7]. None of the multiprocessor scheduling approaches has been found to dominate the other over the complete spectrum of possible application scenarios [1]. Considering the current automotive practice, we expect that automotive multicore scheduling will take an evolutionary approach extending the current fixed priority scheduling towards partitioned fixed priority scheduling.

The classical assumption for real-time tasks in scheduling algorithms is that they are independent. In practice however, tasks need to synchronize the shared resources. When mapping such task systems to multicore architectures, new types of arbitration conflicts emerge that require enhanced scheduling and analysis algorithms. There are two main categories of synchronization mechanisms [9]: *non-blocking* synchronization with *lock-free* execution and *wait-free* execution and *blocking / lock-based* synchronization which can be performed either with *suspending* (when a task suspends, the processor will become available for other work) or with *spinning* (tasks perform a busy-wait until the lock of the required resource is released; in this time the processor and interconnect can not be used by other tasks).

Traditionally, safe sharing and communication in automotive task systems uses OS supported locking and synchronization mechanisms [19, 4]. Thus, lock-based synchronization policies represent the viable solution in order to maintain the compatibility with the current practice.

A locking protocol for multiprocessor real-time systems which fulfills this requirement is the multiprocessor priority

ceiling protocol (MPCP), which is an extension of PCP in the sense that it reduces to PCP when used on single processor systems [20]. Other shared resource arbitration algorithms for multiprocessor systems are also available, for example the multiprocessor stack resource protocol (MSRP) [12] or the flexible multiprocessor locking protocol (FMLP) [8]. Depending on the assumptions on which these protocols rely on, they can be used under either partitioned or global scheduling, with static or dynamic task priority assignments, and where resources are protected by spin-based or suspension-based locks.

Empirical evaluation of the synchronization alternatives for multiprocessor systems under different global and partitioned scheduling algorithms are presented in [9, 10]. From the perspective of efficiency, MPCP may in some setups not be the best choice when compared to other shared resource arbitration methods. An empirical study [9] suggests that non-suspending (spin-based) protocols or lock-free algorithms are often a more efficient choice, in particular when critical sections are short and thus the waiting time is less than the cost of blocking and resuming the process. But spin-waiting increases the system load, with performance being lost due to continuously trying to acquire the lock and consuming communication bandwidth. The more frequently a lock is polled, the sooner will it be acquired, but the more load will be imposed on the arbiters and interconnect.

A critical aspect of efficient system design is the accuracy of the available analyses. The original method to derive response times for tasks with shared resources arbitrated according to MPCP [20] provides only inaccurate upper bounds on the resulting blocking times. More severely, the analysis is constrained to the case of purely periodic task activations with task deadlines smaller than their periods. Real-time theory and practice has since moved far beyond such constrained models. In [18] a first formal analysis method was introduced to analyze a particular evolutionary multicore extension which assumes a partitioned fixed-priority scheduling, the synchronization protocol MPCP and allows for arbitrary task activations. The analysis relies on an improved shared resource load and timing model allowing for more accurate but still safe results and has a wider range of applications matching the current industrial requirements. While [18] (together with [23]) provides a sound formal analysis model, the present paper highlights the key trade-offs in the specification of a shared resource protocol.

2. MULTICORE SYSTEM MODEL

In order to apply analysis techniques known from the research to automotive systems, their underlying analysis models have to fit the automotive structures we find in industry today. For example, in AUTOSAR [4] the “software implementations of functions” are called *software components (SW-C)* that communicate through their

interfaces. The internal behavior of the software components is determined by so called *runnables* of sequential code, that is, procedures that can be executed in response to *events* (such as periodic activation). A common assumption is that runnables inside the software components are mapped to *tasks* for OS scheduling and tasks inherit the activation model of the component runnables [24]. In this way, automotive system models can be adapted to suit the models assumed by the performance and scheduling analysis research methods. For the scope of our paper we assume that runnables are already mapped to tasks and we focus on the tasks timing in multicore setups.

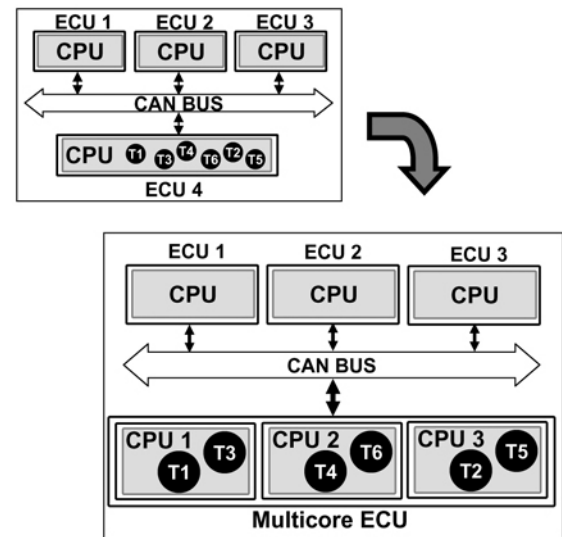


Figure 1. Simplified automotive system with three single-core ECUs and one multicore ECU connected to a CAN bus.

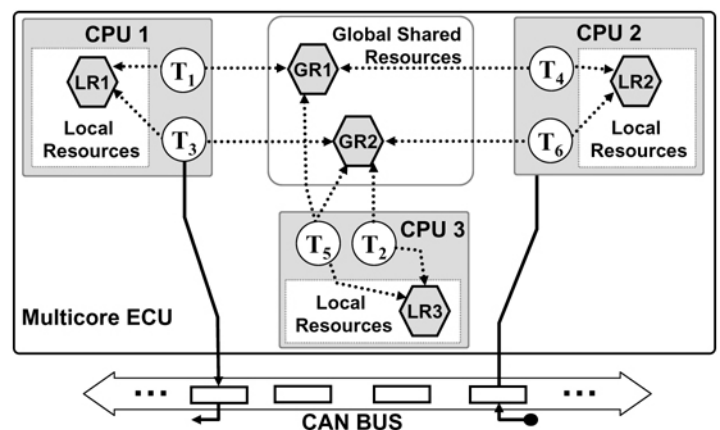


Figure 2. Detailed view of the multicore ECU with tasks accessing local and global shared resources.

Figure 1 shows a possible evolution of a simplified automotive subsystem towards multicore architecture. The initial subsystem is composed of four single-core ECUs and a

CAN bus. In the multicore setup one of the single-core ECUs is replaced by a multicore ECU, for example in order to be able to accommodate further functions or to distribute the load over several cores. In the present paper, we are particularly interested in the behavior of a multicore component, which itself consists of several processors and a set of shared resources that can be accessed from each of the processors as shown in [Figure 2](#).

In this setup, tasks which have to run on the multicore ECU have static priorities, are considered statically mapped to the available processors with some method (e.g. manually or according to a bin-packing heuristic [17]) and cannot migrate at runtime. Also, the set of tasks is assumed to be static and tasks cannot be created dynamically. Two tasks are called local to each other if they reside on the same core and remote if they reside on different cores. On each core of the multicore ECU resides an independent static priority preemptive (SPP) scheduler according to which tasks are locally scheduled. The other components in the system (such as the bus, and the other remote components) may be scheduled arbitrarily.

Each task is activated by events, which can be either external (such as interrupts), or the result of another task or bus communication being finished (in which case there is a partial order between possible task activations). Each time a task is activated it needs to execute for a time no longer than the *worst-case execution time* C_i , and has to complete its execution until a given (relative) *deadline* D_i , which may be smaller, equal or larger than the distance to the successive activation. Thus, if a task has a worst-case response time larger than this distance, it is possible that another instance of this task may be activated before the previous one has completed. In this case, we assume that new instances may not actually start to execute before the preceding has completed and this queuing time will be considered as part of the task's response time.

Shared resources in the system are objects that require serialized access. Each processor has its own *local resources* (LR) which can be used only by the tasks that are mapped on it. Accesses to local resources are arbitrated according to the priority ceiling protocol (PCP). Additionally, in multicore setups, tasks mapped on distinct cores may have to share some common resources. These are called *global resources* (GR), are assumed in a separate shared resource module and can be accessed by tasks mapped to different processors. Accesses to global shared resources are arbitrated according to a lock-based arbitration policy (e.g. the multiprocessor priority ceiling protocol (MPCP) [20]). If a task T_i , running on a core of the multicore ECU as depicted in [Figure 2](#), performs an access to a resource X it will perform a call `GetResource(X)`, e.g. `GetResource(LRx)` or

`GetResource(GRx)`. An occupied resource will be released by a task with a call `ReleaseResource(X)`.

For the next explanations we consider the subsystem depicted in [Figure 2](#) that may be part of a larger automotive system. This subsystem contains a set of hard real-time tasks $T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$, where tasks are numbered in the order of their priority (with T_1 having the highest). Task T_6 on core CPU2 is activated by incoming messages that arrive via a frame on the CAN bus that also services other communications and is arbitrated according to the priority-based CAN protocol [22]. Each of the three CPUs of the multicore ECU are running two tasks which make use of local shared resources according to the OSEK/VDX standard [19]. The shared resources which are used by tasks that are mapped to different cores are the two global shared resources GR_1 and GR_2 .

Relying on this system model we will further highlight the impact the assumptions regarding the scheduling and the resource arbitration policy may have on the systems timing.

3. TIMING IMPLICATION OF MULTICORE COMPONENTS

In single processor ECUs conflicts due to shared resource usage are arbitrated according to the priority ceiling protocol (PCP) and have only a local impact on the timing of the tasks running on the same ECU. In the case of multicore systems, the use of physically shared hardware (e.g. shared memory), or synchronization via logical resources (e.g. semaphores) introduces a global effect and generates dependencies between the task execution on the different cores. The local execution of a task on a core is now influenced by the local execution of other tasks on other cores, thus challenging the real-time behavior of the entire multicore ECU and with this the expected benefits of the multicore setups.

Key requirements for reliable and predictable timing behavior of multicore systems with shared resources are: the support for mutual exclusion between tasks on the same and on different cores; the absence of deadlocks; the absence of unbounded blocking, for example caused by priority inversion; the presence of an upper bound for the tasks' blocking time; and the minimization of this upper bound. Several key aspects which have to be considered by systems' designers in order to fulfill these requirements will be next discussed.

A. LOCAL BLOCKING STRATEGY

According to the specifications of the lock-based arbitration policies, a task T_i which attempts to lock a shared resource will receive the lock if the resource is not already occupied by another task T_j (which can be local or remote with T_i). If

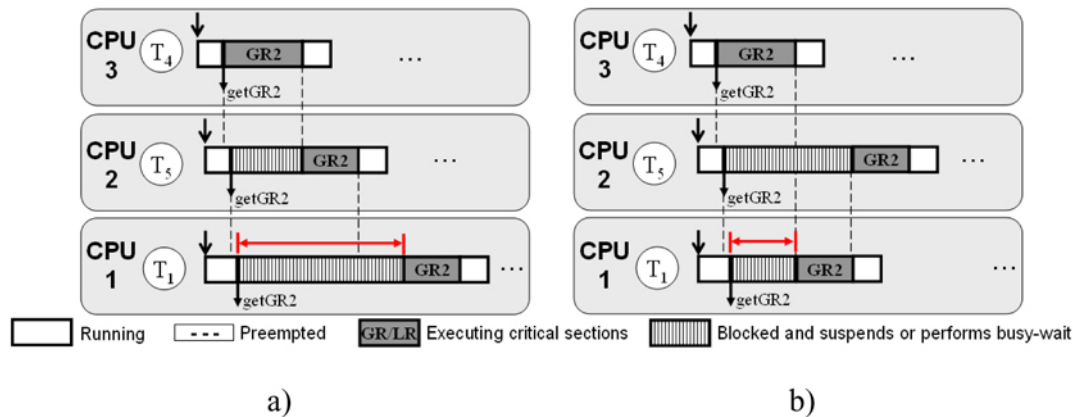


Figure 3. Granting resources in a) FCFS manner and b) priority-based manner when tasks on different cores attempt to lock the same global shared resource.

the resource is already occupied, the task trying to lock the resource will be blocked and either (i) suspends (allowing other local tasks to run on the host core) or (ii) performs a busy-wait thus keeping the local core occupied until it receives the required resource (in case of spinning-based locking protocols).

As the execution of the tasks and therewith the timing of the requests for shared resources initiated by tasks during their execution depend on the scheduling policy, the arbitration of conflicting accesses for resources is related to the scheduling decisions. Thus, the local waiting strategy has an impact on the amount of parallel requests from the processor: Allowing other tasks to execute also gives them the opportunity to request a lock. While this may have a beneficial impact on the tasks finishing time, it also introduces problems like additional priority inversion, and increased load on the shared resource (which in multicore systems also impacts the tasks on the other processors), which need to be considered by the blocking time analysis.

B. ORDER OF GRANTING THE LOCKS

In the case of multiple coinciding requests to the shared resources the lock arbitration policy has to decide the order in which tasks will access the required resources. A possible solution for granting resources is the first-come-first-served (FCFS) arbitration strategy. This method is simple to implement, but counters the prioritization of tasks on their processors. In Figure 3a, where the resources are granted in a FCFS manner, a high priority task (in this case T_1) may be blocked by all other tasks that are using the same global shared resource. This may lead to unacceptable long blocking times for high priority tasks. To avoid such situations the order of granting accesses on the resources has to be correlated to the tasks priorities, similarly to the approach specified by the single-processor priority ceiling protocol (PCP). In case of priority based resource arbitration policies (Figure 3b) task priorities are preserved and the possible

blocking for higher priority tasks is reduced (as considered by the MPCP [20]).

<figure 3 here>

C. SHARED RESOURCE CEILING PRIORITIES

To highlight the benefit of using priority ceilings in multicore setups consider the example in Figure 4a. According to the static-priority preemptive scheduling approach, the higher priority task T_2 may preempt the execution of a lower priority local task T_5 which in the example scenario holds the global shared resource GR_2 . If task T_2 will then also try to lock the GR_2 it will without further measure have to wait forever for the resource as task T_5 remains preempted and does not have anymore the chance to release the occupied resource. Such a situation not only influences the core where these tasks are mapped but also the other cores where tasks may also indefinitely wait for the global shared resource GR_2 (for example task T_3 on CPU1 and task T_6 on CPU3).

<figure 4 here>

Situations similar to the one presented in Figure 4a have been already identified and solved by the priority ceiling protocol (PCP) for single processor systems. The priority ceiling of a local shared resource is defined to be the priority of the highest priority task that may lock that shared resource. A task which locks a shared resource in single-core systems will execute at its assigned execution priority until another task attempts to lock that resource. In this moment the task holding the resource will temporarily raise its execution priority at the priority level of the blocked task and thus will continue executing its critical execution.

A similar approach has to be considered for assigning priority ceilings to global shared resources in multicore setups. Thus,