11AE-0089

AUTOSAR Extensions for Predictable Task Synchronization in Multi-Core ECUs

Karthik Lakshmanan, Gaurav Bhatia, Ragunathan (Raj) Rajkumar
Carnegie Mellon University

ABSTRACT

Multi-core processors are becoming increasingly prevalent, with multiple multi-core solutions being offered for the automotive sector. Recognizing this trend, the AUTomotive Open System ARchitecture (AUTOSAR) standard version 4.0 has introduced support for multi-core embedded real-time operating systems. A key element of the AUTOSAR multi-core specification is the *spinlock* mechanism for inter-core task synchronization. In this paper, we study this *spinlock* mechanism from the standpoint of timing predictability. We describe the timing uncertainties introduced by standard *test-and-set* spinlock mechanisms, and provide a predictable priority-driven solution for inter-core task synchronization.

The proposed solution is to arbitrate critical sections using the well-established Multi-processor Priority Ceiling Protocol [3], which is the multiprocessor version of the ceiling protocol for uniprocessors [1, 2] used by AUTOSAR. We also present the associated analysis that can be used in conjunction with the AUTOSAR task model to bound the worst-case waiting times for accessing shared resources. The timing predictability provided by our protocol is an important requirement for automotive applications from both certification and validation standpoints.

INTRODUCTION

Processor architectures have reached a turning point in their evolution with manufacturers now placing an emphasis on achieving parallelism through multi-cores as opposed to increasing the underlying clock frequency. Excessive power consumption, increasing hardware complexity, clock synchronization problems, and saturation of pipeline optimizations represent reasons for this shift in paradigm towards multi-core processors. Although the introduction of multi-core processors promises a linear increase in raw throughput, it poses many daunting challenges for software development. Existing applications may have to be evolved and parallelized to exploit the available performance from multiple cores. Also, the true parallelism exposed by such architectures violates many assumptions currently made in developing embedded real-time software for single-core processors. Specifically, mutual exclusion protocols such as the single-core priority ceiling protocol [2] and highest-locker priority implementations [7] do not directly apply in the multi-core scenario.

Automotive systems have recently seen a tremendous growth in terms of advanced software features such as driver-assist technologies, active safety systems, and interactive entertainment platforms. Multi-core processors constitute an effective means for realizing such computationally-intensive applications in future automotive platforms. Recognizing these changes in application characteristics and processor capabilities, the AUTomotive Open System ARchitecture (AUTOSAR) Version 4.0 standard has introduced support for multi-core embedded real-time operating systems. New concepts such as locatable entities (LEs), multi-core startup/shutdown, Inter-OS-Application Communicator (IOC), and SpinlockTypes have been introduced in the AUTOSAR multi-core OS architecture specification to extend the single-core OS specifications.

In this paper, we study the *SpinlockType* mechanism specified in AUTOSAR 4.0 from the perspective of timing predictability. We specifically focus on non-nested spinlocks, which are recommended by the AUTOSAR 4.0 specification[1] [1]. Given the context of non-nested spinlocks, we review the sources of unbounded timing introduced by the current specification of AUTOSAR *SpinlockType*: (i) deadlocks, and (ii) starvation. We show example scenarios to illustrate both these problems, and discuss priority-driven approaches to solve them. Based on these discussions, we develop our solution where critical sections can be arbitrated using the well-established Multi-processor Priority Ceiling Protocol. This latter protocol is the multiprocessor version of the ceiling protocol for single-core processors used by AUTOSAR. This solution guarantees bounded delays to accessing critical sections, and avoids the timing unpredictability issues introduced by the current specification of AUTOSAR *SpinlockType*.

[1]AUTOSAR Specification of Multi-Core OS Architecture V1.0.0 R4.0 Rev 1 Section 7.3.29:*"To avoid deadlocks it is not allowed to nest different spinlocks. Optionally, if spinlocks shall be nested, a unique order has to be defined."*

ASSUMPTIONS AND SYSTEM MODEL

We first provide the context for our work by describing the underlying assumptions and our system model. In this work, we specifically focus on basic tasks as defined in AUTOSAR 4.0. As defined by the standard, basic tasks do not block by themselves or wait on any OS events. We denote a basic task executing in the system as τ_i. Each task τ_i is assumed to be a periodic task, which implies that each task repeats itself after a *time frame* or *period* denoted by T_i. Each occurrence of the task τ_i is assumed to require no more than C_i units of execution time on its processor. Each occurrence of the task τ_i is assumed to have a relative deadline equal to the task period, i.e. each occurrence of τ_i should complete before the next occurrence of τ_i. Any task τ_i in the system can therefore be represented by the tuple (C_i, T_i). We define the task set as a collection of n tasks $\{\tau_1, \tau_2, ..., \tau_n\}$ ordered in the increasing order of task periods. We also assume a priority-driven OS scheduler with task priorities assigned in a rate-monotonic fashion, i.e. tasks with shorter periods (and relative deadlines) have higher scheduling priorities [9]. We use the notation $hp(\tau_i)$ to denote the set of all tasks with higher priority than τ_i, and $lp(\tau_i)$ to denote the set of all tasks with lower priority than τ_i. For simplicity of presentation, we will assume that all tasks have unique priorities. We will represent the m processor cores in the system as $\{P_1, P_2, ..., P_m\}$.The priority of task τ_i is denoted as π_i. We will also use the AUTOSAR convention with a higher value of π_i denoting a higher priority.

In this paper, we are interested in tasks that share resources among each other. We will use the notation R_k to denote the k^{th} shared resource. The lock protecting the shared resource R_k will be denoted as S_k. The maximum time for which resource R_k can be held is denoted by $C(R_k)$. We will now leverage this system model in the context of AUTOSAR to provide response-time bounds when resources can be shared across tasks executing on different cores.

AUTOSAR SPINLOCKS

Mutual exclusion in single-core processors is facilitated by AUTOSAR using a function called `GetResource()`. This function leverages the priority ceiling protocol: when a task acquires a resource, its priority is temporarily promoted to the highest priority among all the tasks that can use the resource. This bounds and reduces the duration for which any task waits for a lower-priority task to release a shared resource. In other words, it bounds and significantly reduces priority inversion. Priority inversion cannot be completely eliminated when logical resource sharing is required, but it can be minimized by using appropriate protocols. In the current context, priority inversion cannot be eliminated since the lower-priority task holding the resource has to still release it before the higher-priority task can proceed due to the non-preemptive and mutually-exclusive nature of shared resources. However, `GetResource()` uses the priority ceiling protocol, which ensures that the priority inversion is no more than the duration of a single resource-holding duration. This mechanism does not scale to multi-core processors since priorities are insufficient in preventing access from tasks executing on other cores.

A key extension in the AUTOSAR 4.0 multi-core specification is a new mechanism for mutual exclusion across tasks on different cores called as the *SpinlockType*. This is a busy-waiting mechanism that polls a (lock) variable until it becomes available using an atomic *test and set* functionality. Once a lock variable S_l is obtained by a task, other tasks on other cores will be unable to obtain the lock variable S_l, effectively ensuring the constraint of mutually exclusive access to the resource R_l protected by the lock S_l. This mechanism works with multi-core processors, since it relies on shared memory locks as opposed to the priority attribute that does *not* span cores.

The *SpinlockType* is an approach to address the shortcomings of the priority ceiling protocol when extending AUTOSAR for use in multi-core processors. However, it introduces two problems from a timing standpoint: (i) deadlocks, and (ii) starvation. We will now describe these problems and provide priority-driven approaches to avoid them and achieve response-time guarantees.

DEADLOCKS

In the context of non-nested spinlocks, the *SpinlockType* mechanism could potentially lead to deadlocks, as noted in the standard specification itself:

Deadlock due to preemption

The deadlock happens when a lower priority task holding a resource protected by *SpinlockType A* gets preempted by a higher priority task that later tries to acquire the same *SpinlockType A* (see Figure 1).

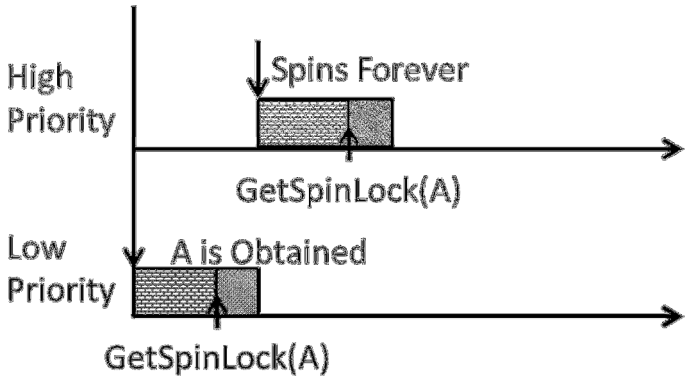

Figure 1 –Deadlock due to Preemption

The current solution presented in the AUTOSAR specification is to (a) return an error to TASK/ISR2 trying to acquire a spinlock assigned to another TASK/ISR2 on the same core (per MCOS0112 and MCOS0113 of the specification), or (b) protect a TASK by wrapping the spinlock with a `SuspendAllInterrupts() call` so that the task cannot be preempted.

With respect to (a), returning an error to the second task trying to acquire the spinlock A is not quite useful. It would be desirable to avoid this scenario in the first place, since application developers should not have to worry about such errors resulting from operating system scheduling decisions.

The advantage of (b) is that using `SuspendAllInterrupts()` reduces the amount of *remote blocking* suffered during multi-core synchronization. In the context of multi-core mutual exclusion, *remote blocking* is defined as the duration for which a task waits for a shared resource to be released on a remote core. Consider an example with three tasks τ_1,τ_2, and τ_3 in decreasing order of task priorities. Let tasks τ_2 andτ_3 be assigned to core P_1, and task τ_1 be assigned to core P_2. Tasks τ_1 and τ_3 share a resource R protected by a *SpinlockType S*. Figure 2 shows the scenario when task τ_3 acquires the resource R and gets preempted by task τ_2 on core P_1 before releasing R. In this case, task τ_1 executing on core P_2 might try to acquire the resource R and get blocked indirectly by τ_2. This remote blocking could be potentially very large if there are many tasks with higher priority than τ_3 on core P_1.

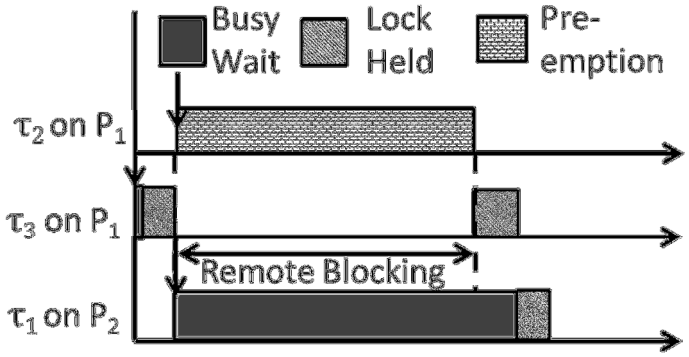

***Figure 2 – Remote Blocking without* `SuspendAllInterrupts`**

The use of `SuspendAllInterrupts()` prevents any preemption for the duration of time in which the shared resource is held. In this example the use of `SuspendAllInterrupts()` leads to a significant reduction in the remote blocking suffered by τ_1 as shown in Figure 3. The caveat here is that the use of `SuspendAllInterrupts()` converts the duration for which a shared resource R is held into a non-preemptible section, which in turn could block other shared resources (say R') potentially required by tasks with higher priority than all tasks that access R. This is the main disadvantage of using recommendation (b) since any high priority task τ_0 requiring shared resource R' and having higher priority than both τ_2 and $\tau_3$will also suffer from the non-preemptive blocking due to R.

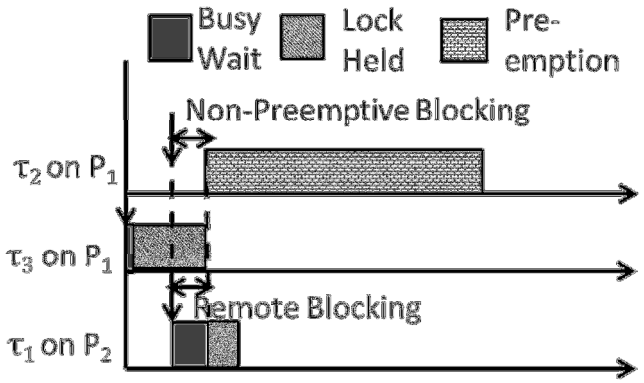

Figure 3 – Remote Blocking with `SuspendAllInterrupts`

The second source of unbounded timing with AUTOSAR *SpinlockType* arises due to starvation as we will describe next.

STARVATION

Consider 3 tasks τ_1,τ_2, and τ_3, each running individually on three cores P_1, P_2, and P_3. All three tasks access the same shared resource R, which is protected using AUTOSAR Spinlocks. In this scenario, we show that task τ_3 can suffer from starvation. As can be seen in Figure 4, depending on the hardware implementation of the *test-and-set* mechanism and the task set characteristics, it could lead to starvation with task τ_3 potentially never getting the Spinlock. In this scenario, the *test-and-set* hardware implementation may not *guarantee* that the *test-and-set* request from core P_3 will succeed before the *test-and-set* requests from cores P_1 and P_2. Even if such a guarantee is provided, there are still scenarios where task τ_3 might be starved as we will show next.

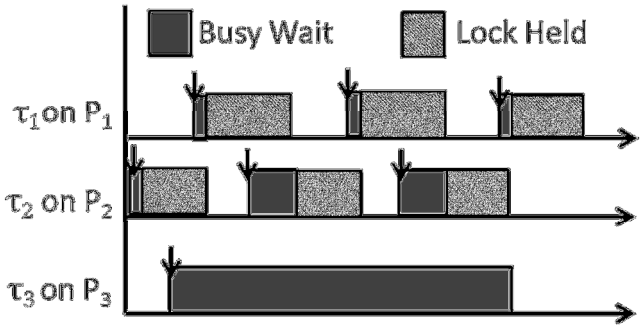

Figure 4 – Task τ_3 suffers starvation with Spinlocks since it can in principle always lose the lock contest.

Another possible scenario is shown in Figure 5, where task τ_1 runs on core P_1 and tasks τ_2 & τ_3 run on core P_2. τ_1 & τ_3 share a resource. In the sequence presented, task τ_3 gets preempted by τ_2, first before the shared resource is released by τ_1 on core P_1. In effect, τ_3 would be starved of the shared resource even though it is available periodically. This scenario is independent of the *test-and-set* implementation.

A bound on the waiting time for the spinlock can be obtained if (A) both the `GetSpinLock()` mechanism used to acquire the spinlock and the actual duration of holding the spinlock itself are wrapped with the `SuspendAllInterrupts` setting. (B) When there are $m(S)$ contenders for spinlock S that can be held for $C(S)$ units of time, then each task issues no more than one request for the spinlock every $m(S)C(S)$ units of time . Condition (A) prevents any preemption when spinning for the lock, thus preventing the scenario described in Figure 5. Condition (B) ensures that the hardware *test-and-set* implementation does not result in starvation as illustrated in the example given in Figure 4.

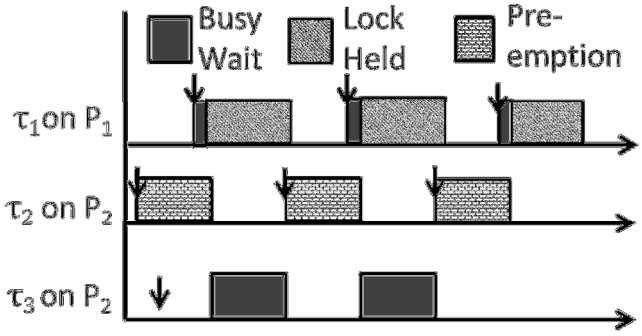

Figure 5 – Starvation with Spinlocks due to Preemption

Although extending the AUTOSAR 4.0 *SpinlockType* mechanism with conditions (A) and (B) will enable us to achieve bounded timing properties for spinlocks, the priority inversion resulting from non-preemptible sections can be quite large, and this can be an especially high overhead for high-priority tasks with really short periods and deadlines. We next describe the multiprocessor priority ceiling protocol. This protocol is an extension of the priority ceiling protocol used in AUTOSAR for single-processor mutual exclusion that provides bounded timing properties, while reducing the priority inversion and utilization loss from remote blocking.

MULTIPROCESSOR PRIORITY CEILING PROTOCOL

The multiprocessor priority ceiling protocol (MPCP) was developed in [3] for specifically dealing with the mutual exclusion problem in the context of shared-memory multiprocessors. Modern multi-core processors largely resemble shared-memory multiprocessors. Thus, MPCP is also a good fit for mutual exclusion in multi-core processors. In the context of MPCP, a *global mutex* is defined as a mutual exclusion lock shared across tasks executing on different cores. A *local mutex* is defined as a mutual exclusion lock that is only shared across tasks executing on the same core. A brief description of MPCP from [3] is applicable in the AUTOSAR context as presented next.

MPCP WITH AUTOSAR SPINLOCKS

- Tasks use their assigned scheduling priority unless holding a mutex.
- The single-core priority ceiling protocol is used for all requests to local mutexes.
- The priority ceiling of a global mutex M is defined as $\pi(M) = \pi_G + \pi_c$, where π_G is a priority higher than all normal scheduling priorities assigned to tasks in the system, and π_C is the highest normal priority of any task accessing M.
- Any task holding a global mutex M executes at the priority ceiling of global mutex M.
- Any task holding a global mutex M_1 can preempt another task holding a global mutex M_2, if the priority ceiling of M_1 is higher than the priority ceiling of M_2.
- When a task τ requests a global mutex M, M can be granted by means of an atomic transaction on shared memory, if M is not already held by any other task.
- If a request for a global mutex M cannot be granted, then the requesting task τ is added to a prioritized queue on M. In a suspension-based implementation, task τ will be blocked on the event that M is released. In a spinning-based implementation, task τ will spin on a local variable until M is released.
- When a task releases a global mutex M, the highest priority task τ waiting in the prioritized queue for M is signaled on its local core, and M is marked as being held by τ. If there are no tasks waiting for M, then it is just marked as released.

ILLUSTRATION

Figure 6 illustrates an example scenario where MPCP is applied. There are 3 tasks τ_1, τ_2, and τ_3, with tasks τ_1 and τ_2 executing on core P_1 and task τ_3 executing on core P_2. Tasks τ_2 and τ_3 share a resource R, which is arbitrated using the Multi-processor Priority Ceiling Protocol with a global mutex M.

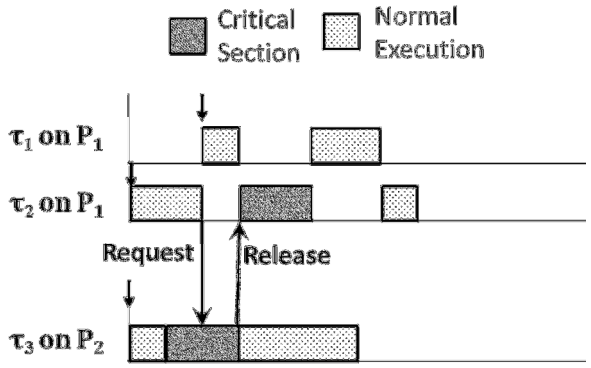

Figure 6 – Illustration of MPCP

In this example scenario, task τ_2 starts executing on processor P_1 while task τ_3 starts executing on processor P_2. Task τ_3 acquires the mutex M and hence gets exclusive access to resource R. Meanwhile, task τ_2 requests the resource R and gets blocked on mutex M. When task τ_1 gets released, it continues to execute. Whenever the mutex M is released on processor P_2 and it is given to task τ_2, task τ_2 acquires a priority greater than all normal executing priorities thereby preempting task τ_1. When τ_2 releases the mutex M, it reverts back to its normal priority thereby enabling τ_1 to preempt τ_2 after the release of M.

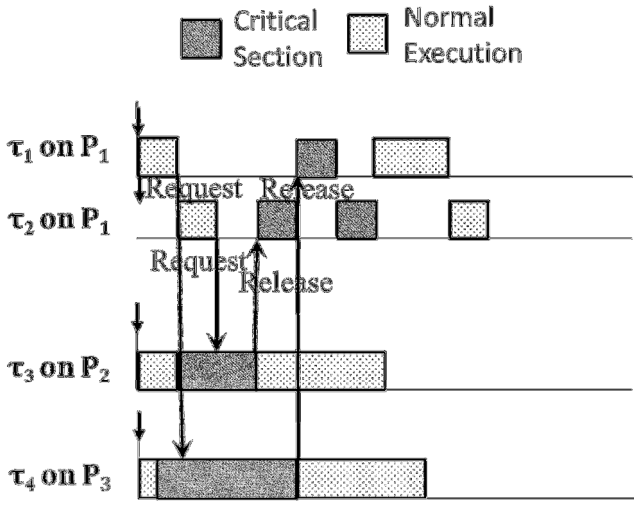

Figure 7 – Preemption of Critical Sections under MPCP

A perhaps more interesting scenario is illustrated in Figure 7. In this example, there are 4 tasks τ_1, τ_2, τ_3 and τ_4, with tasks τ_1 and τ_2 executing on core P_1, task τ_3 executing on core P_2, and task τ_4 executing on core P_3. Tasks τ_2 and τ_3 share a resource R_1, which is arbitrated using the Multi-processor Priority Ceiling Protocol with a mutex M_1. Tasks τ_1 and τ_4 share a resource R_2, which is arbitrated using the Multi-processor Priority Ceiling Protocol with a mutex M_2.

For the scenario in Figure 7, all the tasks are released simultaneously in their respective processor cores. Tasks τ_1 preempts task τ_2 since it has a higher scheduling priority. First, task τ_4 acquires the resource R_2 by locking the mutex M_2. Task τ_3 then acquires the resource R_1 by locking the mutex M_1. When task τ_1 tries to acquire R_2, it gets blocked on mutex M_2. Task τ_2 therefore starts executing on processor P_1 while task τ_3 continues executing on processor P_2. Later, task τ_2 requests the resource R_1 and gets blocked on mutex M_1. When mutex M_1 is released on processor P_2 and it is given to task τ_2, task τ_2 acquires a priority greater than all normal executing

priorities. However, when the mutex M_2 is released on processor P_3 and it is given to task τ_1, this mutex M_2 has a higher global priority ceiling than the mutex M_1 since M_2 is shared between τ_1 and τ_4, while M_1 is shared betweenτ_2 and τ_3. Task τ_1 therefore preempts task τ_2 that is holding mutex M_1. This preemption ensures that the mutex M_2 required by a task with higher priority does not get blocked by mutex M_1 shared among lower priority tasks. This preemption is not possible when using the standard AUTOSAR *SpinlockType* with `SuspendAllInterrupts()`. MPCP thus provides a priority-driven approach to dealing with shared resources, as opposed to the AUTOSAR *SpinlockType*, which would effectively lead to non-preemptive sections.

From an implementation perspective, the prioritized queues can be located in shared memory. The task releasing the resource is responsible for signaling the highest-priority task waiting on the priority queue. If no task is waiting on the priority queue, then the task can simply mark the resource as available. The latency of locking this priority queue is dependent on the processor implementation and the memory system.

RELATED WORK

The mutual exclusion problem in the context of single-core processors and static priority scheduling was addressed in [2]. The proposed solution known as the Priority Ceiling Protocol (PCP) has been adopted in AUTOSAR for mutual exclusion in single-core processors. For dynamic priority scheduling systems, an alternative mechanism known as the Stack Resource Policy (SRP) was proposed in [8]. These mutual exclusion techniques have been specifically designed for the single-core context, and do not extend directly to the multi-core scenario. As noted by the AUTOSAR 4.0 [1] specification, the key underlying problem here is that priorities are effective in preventing access to shared resources in a single-core context but do not work across multiple cores. The AUTOSAR 4.0 specification therefore proposes the *SpinlockType* mechanism for handling mutual exclusion in multi-core processors. The contributions of our work in this regard are (i) highlighting the timing issues such as deadlock and starvation that can occur when using *SpinlockTypes*, and (ii) developing extensions and associated analysis for achieving predictable timing with *SpinlockTypes*.

The issue of extending PCP to the multiprocessor context was considered in [3], and the multiprocessor priority ceiling protocol (MPCP) was proposed. MPCP was designed for shared memory multiprocessor systems, and also applies to multi-core processors which have similar architectural characteristics. In this work, we have reviewed MPCP in the context of AUTOSAR *SpinlockTypes* to reduce priority inversion and contain the utilization loss resulting from remote blocking. We have also leveraged the analysis provided in [6], which considers both suspension and spinning based versions of the MPCP protocol. The key contribution of our work in this regard is to adapt the existing MPCP results in the AUTOSAR context, and provide the required timing analysis.

A flexible approach for mutual exclusion was studied in [4], where short resource requests use a busy-wait mechanism, while long resource requests are handled using a suspension approach. There are two key differences between our MPCP-based approach and the one used in [4] for long resource requests: [4] uses a First-In-First-Out queue for determining the task that gets the resource, whereas MPCP uses prioritized queues, and [4] uses the standard priority ceiling for long request locks, whereas MPCP uses a global priority ceiling that is higher than all normal execution priorities and reduces *remote blocking*. Under many conditions [6], the MPCP-based approach performs better.

In the Appendix, we capture the analyses that can be performed for the various schemes we have discussed.

SUMMARY/CONCLUSIONS

Multi-core processors are becoming increasingly prevalent in the general-purpose computing market. Chip vendors in the embedded market have also started offering multi-core chips due to their power and performance benefits. These trends have influenced the AUTOSAR 4.0 release to add support for multi-core processors. The proposed multi-core extension identifies that the single-core priority ceiling protocol does not directly extend to multi-core processors, and introduces a new *SpinlockType* mechanism for mutual exclusion when resources are shared across cores. In this work, we studied different sources of unbounded timing such as deadlock and starvation introduced by the *SpinlockType* mechanism as currently defined in AUTOSAR 4.0. We then discussed extensions to achieve timing predictability with the *SpinlockType*. Subsequently, we have developed our approach where resources can be arbitrated using the Multi-processor Priority Ceiling Protocol (MPCP). The resulting solution provides bounded timing support, thus it is beneficial from both certification and validation standpoints.

REFERENCES

1. AUTOSAR 4.0 Multi-Core OS SWS – Specification of Multi-Core OS Architecture.

2. L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, pp. 1175-1185, September, 1990.
3. R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on* , vol., no., pp.116-123, 28 May-1 Jun 1990.
4. Block, A.; Leontyev, H.; Brandenburg, B.B.; Anderson, J.H.; , "A Flexible Real-Time Locking Protocol for Multiprocessors," *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, vol., no., pp.47-56, 21-24 Aug. 2007.
5. Brandenburg, B. B., Calandrino, J. M., Block, A., Leontyev, H., and Anderson, J. H.,"Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?," In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium* (April 22 - 24, 2008). RTAS. IEEE Computer Society, Washington, DC, 342-353.
6. Karthik Lakshmanan, Dionisio de Niz, Ragunathan Rajkumar, "Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors," Real-Time Systems Symposium, IEEE International, pp. 469-478, 2009 30th IEEE Real-Time Systems Symposium, 2009.
7. R. Rajkumar, "Synchronization in Real-Time Systems", Ph.D. Thesis, Carnegie Mellon University, 1989.
8. Baker, T. P, "Stack-based scheduling of real-time processes", Real-Time Systems, 1991, Springer, vol. 3, issue 1, pp. 67-99.
9. Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", J. ACM 20, 1, Jan. 1973, pp. 46-61.

CONTACT INFORMATION

Karthik Lakshmanan: klakshma@andrew.cmu.edu

Gaurav Bhatia: gnb@andrew.cmu.edu

Prof. Ragunathan Rajkumar: raj@ece.cmu.edu

ACKNOWLEDGMENTS

The authors would like to thank Paolo Giusto and Haibo Zeng from General Motors (GM) for their extensive support with the problem formulation, and valuable guidance in identifying the practical solutions.

DEFINITIONS/ABBREVIATIONS

PCP	Priority Ceiling Protocol
MPCP	Multiprocessor Priority
RMS	Rate-Monotonic Scheduling
ISR	Interrupt Service Routine
OS	Operating System
WCET	Worst-Case Execution Time

APPENDIX: ANALYSIS

When tasks do not share any resources, rate-monotonic scheduling analysis can be used to derive the response time of each task occurrence, and determine whether all the tasks in the system are guaranteed to meet their deadlines. Under this analysis, the worst-case response time W_i of task τ_i assigned to a core P_k is given by the following convergence equation [7]:

$$W_i^0 = 0$$

$$W_i^{n+1} = C_i + \sum_{\forall \tau_j | \tau_j \in P_k \& \tau_j \in hp(\tau_i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil C_j \qquad (1)$$

Remote Blocking without SuspendAllInterrupts()

For remote blocking without `SuspendAllInterrupts()`, the worst-case lock holding time $(L_i(S_l))$for any task τ_i for the spinlock S_l is given by:

$$L_i(S_l) = C(S_l) + I_{i,i}(C(S_l)) \qquad (2)$$

Where $I_{i,i}(C(S_l))$ denotes the worst-case preemption suffered by task τ_i when holding the spinlock S_l at its normal scheduling priority.

Remote Blocking with SuspendAllInterrupts()

For remote blocking with `SuspendAllInterrupts()`, the worst-case lock holding time $L_i(S_l)$ for any task τ_i for the spinlock S_l is:

$$L_i(S_l) = C(S_l) \qquad (3)$$

However, any task τ_h executing on core P_k needs to account for non-preemptive blocking B^n_h from lower-priority tasks given by:

$$B_h^n = max_{\forall S_l | \exists \tau_j, \tau_j \in S_l \,\&\, \tau_j \in lp(\tau_h) \,\&\, \tau_j \in P_k} C(S_l) \qquad (4)$$

Remote Blocking with Standard Single-Processor Priority Ceiling Protocol:

In this case, the worst-case lock holding time $L_i(S_l)$ for any task τ_i for the spinlock S_l is given by:

$$L_i(S_l) = C(S_l) + I_{i,\pi(S_l)}(C(S_l)) \qquad (5)$$

where $\pi(S_l)$ denotes the highest scheduling priority among all tasks that can acquire the spinlock S_l, and $I_{i,\pi(Sl)}(C(S_l))$ denotes the worst-case preemption suffered by task τ_i when holding the spinlock S_l at its priority ceiling of $\pi(S_l)$. Comparing Equations (5) and (2), we see that (5)≤ (2), since the priority ceiling of S_l is higher than or equal to the priority of task τ_i, and hence the interference from higher priority tasks when executing at the priority ceiling will be less.

$$I_{i,\pi(S_l)}(C(S_l)) \leq I_{i,i}(C(S_l))$$

Any task τ_h executing on core P_k must account for non-preemptive blocking B^n_h from lower-priority tasks, which could be holding a spinlock and blocking τ_h from getting scheduled on the processor. This non-preemptive blocking B^n_h is given by:

$$B_h^n = max_{\forall S_l | \exists \{\tau_j, \tau_r\}, \{\tau_j, \tau_r\} \in S_l \& \tau_j \in lp(\tau_h) \& \tau_r \notin lp(\tau_h) \& \tau_j \in P_k} C(S_l) \qquad (6)$$

Comparing Equations (4) and (6), we find that (6) ≤ (4) since (4) considers all lower priority tasks $\tau_j \in lp(\tau_h)$ assigned to processor P_k and acquiring a spinlock S_l, while (6) only considers the lower priority tasks τ_j that hold a spinlock S_l shared with a task τ_r with higher priority or equal priority to τ_h ($\{\tau_j, \tau_r\} \in S_l \& \tau_j \in lp(\tau_h) \& \tau_r \notin lp(\tau_h)$). From a designer's perspective, the conclusion is that the blocking duration under the single-processor priority ceiling protocol is less than or equal to the blocking term from spinlocks.

Waiting Time Bound with SuspendAllInterrupts():

If conditions (A) and (B) are in place, the worst-case waiting time $\omega_i(S_l)$ of a task τ_i for a lock S_l can be bounded, and is given by:

$$\omega_i(S_l) \leq (m(S_l) - 1) C(S_l) \qquad (7)$$

Where, $m(S_l)$ is the number of tasks that share the spinlock S_l.

The duration of the non-preemptible section $N(S_l)$ for spinlock S_l wrapped by the `SuspendAllInterrupts()` call is then given by:

$$N(S_l) \leq m(S_l) C(S_l) \qquad (8)$$

Inequality (1) can be extended for the task τ_i accessing shared resources protected using spinlocks and surrounded (including the `GetSpinLock()`) using `SuspendAllInterrupts()`. We assume that C_i includes the time $C(S_l)$ spent holding the resource S_l but excludes the waiting time for S_l (reasonable if the WCET is defined as truly the worst-case execution time obtained by running the

task in isolation), then the response-time W_i for a task τ_i on core P_k that can acquire a spinlock S_l is given by the convergence with initial condition $W_i^0 = 0$:

$$W_i^{n+1} = C_i + max_{\forall S_t | \exists \tau_j, \tau_j \in P_k \& \, \tau_j \epsilon lp(\tau_i) \& \tau_j \in S_t}(N(S_t)) +$$

$$\sum_{\forall S_l | \tau_i \epsilon S_r} \omega_i(S_l) + \sum_{\forall \tau_j | \tau_j \in P_k \& \tau_j \in hp(\tau_i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + \sum_{\forall S_r | \tau_j \epsilon S_r} \omega_j(S_r)) \quad (9)$$

where, $\omega_i(S_l)$ is the maximum waiting time for the lock S_l when using SuspendAllInterrupts().

Waiting Time Bound with MPCP:

When tasks only use MPCP for synchronization on shared resources, the worst-case response time W_i for a task τ_i executing on a core P_k that can acquire shared resource guarded by spinlocks is given by the convergence with initial condition $W_i^0 = 0$:

$$W_i^{n+1} = C_i + max_{\forall S_t | \exists \tau_j, \tau_j \in P_k \& \, \tau_j \epsilon lp(\tau_i) \& \tau_j \in S_t}(C(S_t)) +$$

$$\sum_{\forall S_l | \tau_i \epsilon S_l} \mu_i(S_l) + \sum_{\forall \tau_j | \tau_j \in P_k \& \, \tau_j \in hp(\tau_i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + \sum_{\forall S_r | \tau_j \epsilon S_r} \mu_j(S_r)) \qquad (10)$$

where, $\mu_i(S_l)$ is the maximum waiting time for the lock S_l when using MPCP (see [6] for a more details regarding MPCP analysis).

The duration $H_i(C(S_l))$for which a critical section of length $C(S_l)$ can be held by a task τ_i executing on processor P_k is given by:

$$H_i(C(S_l)) = C(S_l) + \sum_{\forall \tau_j | \, \tau_j \in P_k} max_{\forall S_t | \tau_j \in S_t \, \& \, \pi(S_t) > \pi(S_l)}(C(S_t))$$

where, $\pi(S_l)$ is the global priority ceiling for lock S_l.

An upper bound $\mu_i(S_l)$ on the waiting time for task τ_i, when trying to acquire lock S_l using a spinning-based version of MPCP, is given by the convergence:

$$\mu_i^0(S_l) = max_{\forall \tau_j | \tau_j \epsilon lp(\tau_i) \& \tau_j \in S_l} H_j(C(S_l))$$

$$\mu_i^{n+1}(S_l) = max_{\forall \tau_j | \tau_j \epsilon lp(\tau_i) \& \tau_j \in S_l} H_j(C(S_l)) + \sum_{\forall \tau_h | \tau_h \epsilon hp(\tau_i) \& \tau_h \in S_l} \left(\left\lceil \frac{\mu_i^n(S_l)}{T_h} \right\rceil + 1 \right) H_h(C(S_l)) \qquad (11)$$

From a practical perspective, Equation (11) leads to higher-priority tasks having shorter waiting times on global mutexes, while Equation (7) leads to a uniform waiting duration across all tasks.

Waiting Time Bound with a Hybrid Approach:

In this case, let ξ denote the set of all spinlocks that are held for a short duration of time and protected using `SuspendAllInterrupts()`, while λ denotes the set of all spinlocks that use MPCP. In this case, the worst-case response time W_i for a task τ_i executing on a core P_k is given by the convergence: $W_i^0 = 0$

$$W_i^{n+1} = C_i +$$

$$\max \left(max_{\forall S_t | \exists \tau_j, \tau_j \in P_k \& \, \tau_j \epsilon lp(\tau_i) \& \tau_j \in S_t \, \& S_t \in \lambda}(C(S_t)), max_{\forall S_u | \exists \tau_j, \tau_j \in P_k \& \, \tau_j \epsilon lp(\tau_i) \& \tau_j \in S_u \, \& S_u \in \xi}(N(S_u)) \right) +$$

$$\sum_{\forall S_l | \tau_i \epsilon S_l \,\&\, S_l \in \lambda} \mu_i(S_l) + \sum_{\forall S_f | \tau_i \epsilon S_f \,\&\, S_f \in \xi} \omega_i(S_f) +$$

$$\sum_{\forall \tau_j | \tau_j \in P_k \& \tau_j \in hp(\tau_i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + \sum_{\forall S_r | \tau_j \epsilon S_r \,\&\, S_r \in \lambda} \mu_j(S_r) + \sum_{\forall S_q | \tau_j \epsilon S_q \,\&\, S_q \in \xi} \omega_j(S_q)) \quad (12)$$