Investigating the Correctness and Efficiency of MrsP in Fully Partitioned Systems

Shuai Zhao and Andy Wellings
Department of Computer Science, University of York, UK
{zs673, andy.wellings}@york.ac.uk

***Abstract*—MrsP is a FIFO spin-based protocol that adopts a helping mechanism, where a resource holder can migrate to a remote processor to keep executing if it is preempted. In practice, allowing resource-holding tasks to migrate can raise implementation issues and run-time corner cases. In this paper, we present an investigation of the correctness and efficiency of implementing MrsP in fully partitioned systems. We identify potential race conditions and corner cases of the protocol due to the use of migrations. Then, new facilities are proposed to prevent the issues and to provide more efficient resource-accessing behaviours. Finally, evaluations are performed to demonstrate the impact of the run-time issues and to testify the effect of proposed facilities.**

I. Introduction

A. Background and Motivation

Resource control technology for multiprocessors has received much attention in recent years to cope with the transition from uniprocessors to multiprocessors. Among the existing protocols, Burns and Wellings proposed the Multiprocessor resource sharing Protocol (MrsP) [7], which adopts a helping mechanism whereby a preempted resource-holding task can migrate to a processor that is executing a task that is spin-waiting for the same resource. With the helping mechanism, this protocol is attractive in theory as the resource holder can keep making progress when preempted.

However, in practice, the realisation of the helping mechanism can be problematic. The migration targets for a resource-holding task are not constant as remote spinning tasks can also be preempted. Thus, the migration target decision made by the protocol may conflict with the scheduling decisions, and thereby results in incorrect and useless migration behaviours. In addition, a resource-holding task could incur frequent preemptions on each migration target, and therefore results in too frequent migrations so that the task spends more time on migrating rather than executing. These issues can cause unpredictable task behaviours with considerable run-time overheads, which directly undermine the efficiency of the protocol.

In this paper we address the above concerns of MrsP in practice. We start by describing the potential race conditions and corner cases when applying MrsP in real-world fully partitioned systems. New mechanisms and facilities are then proposed with design details to provide correct and efficient run-time task behaviour under MrsP. Finally, a set of evaluations are conducted to demonstrate the impact of such migration issues and an improved efficiency of MrsP implementation with proposed facilities.

B. Related Work

Locking protocols on uniprocessor uniprocessor systems have been well practised for years. Among them, the Priority Ceiling Protocol (PCP) [15], Stack Resource Protocol (SRP) [1] and Deadline Floor Protocol [6] are agreed as the best approach, which minimise the blocking time while avoiding deadlocks with low run-time overheads [9]. On multiprocessors, MPCP [14] requires resource-requesting tasks to explicitly migrate to a predefined processor before they get the resource. However, this mandatory migration approach can impose considerable overheads to the system and hence, undermines the performance of the protocol [16]. In MSRP [11], resources are protected by non-preemptive spin locks in a FIFO order, where access to global resources can be granted locally without the need of migrations. In FMLP [2], resources are grouped by their length, where long resources are protected by semaphores and short resources are controlled by FIFO spin locks to achieve a better performance. The notion of helping is also applied in M-BWI [10] and SPEPP [17], where the resource holder can be helped when being prevented from executing. More recent, RNLP [18] is developed to be the first protocol that supports fine grained nested resource access through a token mechanism and request-satisfaction mechanism. In [13], a task partitioning and resource allocating algorithm is proposed to offer a guaranteed speedup.

Besides protocols, the Holistic Analysis by Brandenburg [4] provides a new approach to account for blocking, which is less pessimistic than the approach applied in MSRP's original analysis [11]. Later, Wieder and Brandenburg developed an analysis framework with Integer Linear Programming technique, which provides more accurate and less pessimistic analysis than that of the holistic approach and can be applied to 8 protocols [19]. In practice, [3] and [5] have implemented SRP, PCP, DPCP, MPCP and FMLP into Litmus^{RT} [4] with their performance investigated and compared.

The research addressing MrsP covers both theory and practice. In [21], a new schedulability analysis for MrsP is proposed, including a new migration cost analysis. In [12], a complete approach to support nested resource access in MrsP is presented with sufficient analysis. In [8], Catellani et al.. demonstrated that MrsP can be effectively implemented in RTEMS and provided a simple prototype implementation of

MrsP in Litmus^{RT}. More recently, Shi et al provided a fully functional MrsP implementation in Litmus^{RT} and compared the performance of MrsP, MPCP, DPCP and DNPP (Distributed Non-Preemptive Protocol) [16]. However, both work focus on the functionality of the protocol and does not discuss the potential issues introduced by migrations in MrsP.

II. MrsP

MrsP [7] is a multiprocessor locking protocol for fully partitioned systems with fixed priorities. Under MrsP, spin locks are adopted and resources are served in a FIFO order. However, MrsP defines that tasks should only spin at the local ceiling priority (i.e., they are preemptable) to benefit high priority tasks. In MrsP, each resource has a ceiling priority on each processor that contains tasks requesting it, which is the highest priority among the requesting tasks. Once a task requests a resource, it raises its priority to the local ceiling of the resource and spins if the resource is not free.

With FIFO spin, MrsP sets a fixed bounded length of the waiting queue, which is the number of processors that contain tasks that request the resource. However, spinning at the local ceiling level can lead to a prolonged blocking time as the resource holder can be preempted by higher priority local tasks. To reduce the blocking, a helping mechanism is introduced in MrsP to help the preempted resource holder. The helping mechanism allows the preempted resource holder to migrate to a remote processor with a running task spinning for the same resource. If preempted again, the holder can migration to its initially assigned processor (if the preemptor is finished) or to another valid processor (if any). After releasing the resource, the task migrates back to its designated processor (if necessary).

With the helping mechanism, the holder can keep making progress by using the wasted cycles of the spinning task. In the worst case, a resource-requesting task needs to help all tasks before it in the FIFO queue each time it tries to access the resource, which leads to a worst case blocking time of the length of the FIFO queue multiplied by the cost for accessing the resource.

III. Issues of Migrations in MrsP

The migration target for a resource holder in MrsP is dynamically decided by whether the remote processor has a running task spinning for the same resource. Thus, the migration target identified can become invalid if the spinning task itself is preempted so that the holder migrates but cannot execute at all (i.e., *false migrations*). In addition, as briefly described in [21], a resource holder may be preempted frequently in systems with high priority tasks with short periods, which can lead to *frequent migrations* with considerable overheads.

A. False Migrations

With the generic Linux kernel, task migrations are handled by a set of push and pull operations, as part of the scheduling routine. The push operation is triggered after a scheduling decision to migrate the previous scheduled task (i.e., the task that was executing before this scheduled task) to a remote processor. The pull operation is preformed before a scheduling decision to migrate a remote task to the local processor. According to [4], the fact that both push and pull operations need to manipulate multiple run queues can cause concurrent state changes and it is not possible to have a consistent snapshot without locking all the run queues. Thus, the migration facility in Linux may either trigger superfluous migrations or fail to trigger required migrations due to such race conditions, resulting in unbounded priority inversion. Similar migration failures can occur when adopting MrsP into such a partitioned run queue structure. We identify two major migration problems of MrsP with such push and pull migration operations.

The first migration problem is caused by race conditions between run queues and can happen in both push and pull operations. Once a resource holder is preempted and a migration target is identified, the holder will be placed into the remote run queue. However, before the next scheduling point, a higher priority task can be released immediately so that the migrated task is not considered by the scheduler at all. Such migration can be regarded as a futile attempt as it only provides extra overheads with the need for further migrations rather than offering the task a real chance to execute.

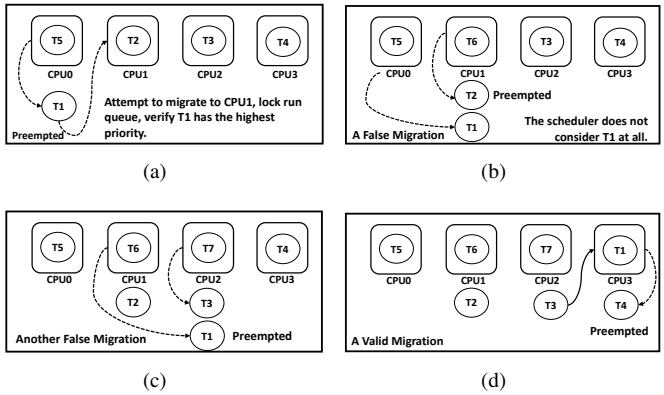

Fig. 1. False Migrations Due to Race Condition.

Figure 1 illustrates this problem with a four core system, where task 1 to 4 request the same resource with low priorities while task 5 to 7 are irrelevant high priority tasks. In Figure 1(a), task 1 (τ_1) is preempted at processor 0 (P_0) while holding the resource so that it migrates to P_1, where τ_2 is spinning for the resource. However, after τ_1 is inserted into the run queue of P_1 (Rq_1), τ_6 is released and is then scheduled to execute. Thus, τ_1 remains in Rq_1 without any chance to execute so that it seeks another processor (Figure 1(b)). In Figure 1(c), the same issue occurs when τ_1 migrates to P_2 so that τ_1 is placed in Rq_2 with no chance to execute. Finally, it migrates to P_3 (Figure 1(d)), where it preempts the spinning task and executes. In this example, 3 migrations are preformed in order to migrate τ_1 to a valid processor, yet two of them are invalid due to immediate updates of run queues.

The second issue is caused by the push operation, which is usually configured with a fixed number of attempts to

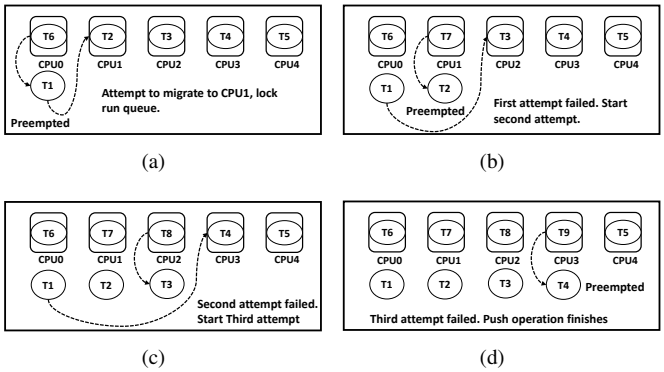

Fig. 2. Missing Necessary Migration due to Limited Attempts.

control overheads. Figure 2 demonstrates this issue with a system of five processors and 3 push attempts. As shown in Figure 2(a), after τ_1 is preempted, the push operation firstly attempts to migrate τ_1 to P_1. However, due to the release of τ_7 in Figure 2(b), the first attempt fails. In Figure 2(c) and 2(d), the second and third attempts fail as well due to the same reason. Thus, the push operation finishes without checking P_4, which is a valid migration target. Such failure can cause a longer resource accessing time of the holder and in consequence, a longer blocking time of all waiting tasks.

Admittedly, a migrated resource holder can be preempted again just after being scheduled, which also requires further migrations. However, false migrations impose extra incorrect behaviours and extra run-time overheads to tasks rather than offering tasks a real chance to execute. In Section V we demonstrate the impact of this issue with experiments.

B. Frequent Migrations

Even if the migrations are correctly performed, the protocol can still be pessimistic due to the helping mechanism. In MrsP, a migrated resource holder can be preempted again so that the task needs to seek further migrations. Thus, in the situation where there exist a large number of migration targets and each of them contains one or more high priority tasks with very short periods, the holder can incur frequent preemptions immediately after being migrated and scheduled. As a result, the resource holder requires more migrations to execute with the resource (revealed by tests in Section V). In addition, it is possible that the holder spends much more time on migrating rather than executing, which greatly undermines the efficiency of the protocol.

IV. Solutions of Migration Issues

To prevent the migration issues identified in Section III, new facilities are introduced for correct and efficient migration behaviours in MrsP. The new facilities are integrated into a MrsP implementation under Litmus^{RT} [4], which provides a real-time testbed for Linux with several pluggable real-time schedulers. The details of the MrsP implementation[1] with new facilities adopted is presented in [20].

A. False-migration-free Mechanism

To avoid false migrations we propose that (1) the helping mechanism should be realised by pull operations only and (2) the migration decisions of the protocol should be made as a part of the scheduling decisions.

With a partitioned run queue structure, the push operation suffers from inescapable race conditions unless obtaining all run-queue locks. As scheduling decisions are made independently on each processor, it is not possible to guarantee that there will not be any release of high priority tasks on the target processor during the migrations by push. In addition, as explained in III-A, necessary migrations can be omitted due to a limited number of attempts. Therefore, push operations should not be adopted for the MrsP implementation to prevent race conditions.

In addition, to prevent race conditions in pull operations, we require that the pull operation needs to be modelled inside the scheduler and as a part of scheduling decisions. During each scheduling point, the pull operation will be triggered if the to-be-scheduled task is spinning for a resource while the resource holder is being preempted on a remote processor. The scheduler then replaces the to-be-scheduled task with the preempted resource holder as the next task to schedule. Thus, the migrated task is always eligible to execute while any newly released high priority tasks need to invoke the scheduler to preempt.

To realise the false-migration-free mechanism, a preemption queue (Pq) and a Pq lock are introduced for each processor. Once a resource-accessing task (either holding or waiting for a resource) is preempted, it will be placed into the Pq of its original processor rather than the Rq of the current processor. Upon a scheduling point, the scheduler looks into its local Pq and Rq and takes the highest priority task to execute. By doing so, the resource accessing task is able to resume on its original processor even though it is preempted on a remote processor. In addition, if the to-be-scheduled task is waiting for a resource while the resource-holding task is preempted (i.e., being placed into Pq), the pull operation removes the task from the Pq and migrates it to the resource-waiting task's processor to execute. To avoid race conditions, the Pq lock must be obtained in order to access that Pq.

By adopting such a facility, we realise the required functionalities defined in the helping mechanism. Meanwhile, we can avoid accessing multiple run queues with the nested access of Rq locks. As the lock of the Pq needs to be acquired inside the scheduler, i.e., after obtaining the Rq lock, deadlocks are prevented because no circular access can be formed. Yet it seems that the cost for a scheduling decision can be increased as the scheduler may need to compete for the Pq locks. However, such competition only occurs if a scheduler

[1]The implementation can be accessed online at https://github.com/RTSYork/MrsP_Implementation_Litmus.

is trying to pull a preempted holder (i.e., the to-be-scheduled task is waiting for a resource). Hence, in the viewpoint of cost, there is no difference between spinning for the resource or spinning for a Pq lock to offer help. With the support of the false-migration-free mechanism, we eliminate possible race conditions between processors while migrating so that each migration is a valid migration: the resource holder is guaranteed a chance to execute after migrated. In Section V-B, the evaluation result demonstrates that such a "false-migration-free" implementation is important to the usability of the protocol.

B. Non-Preemptive Sections

To avoid frequent migrations of a resource holder and to improve the efficiency of the helping mechanism, we integrate MrsP with a short non-preemptive section (NP section) to offer a trade off between the maximum number of migrations a holder can suffer and bounding the resulting blocking time on high priority tasks. Upon each migration, the resource holder is allowed to execute non-preemptively for a short period before it inherits the ceiling priority on the current partition. Accordingly, any newly released high priority tasks have to cope with the cost of one NP section before it can preempt the holder and execute. In this paper we set the length of the NP section to double the cost of migration. However, such a parameter can be tuned as long as the high priority tasks are able to meet their deadlines.

With NP sections, a migrated resource-accessing task will be assigned with the priority 0 (which is reserved by Litmus^{RT} for priority boosting) so that it can execute effectively non-preemptively. To restore the corresponding ceiling priority of the task after the NP section, one high resolution timer (`hrtimer`) is introduced for each processor. The `hrtimer` will be set each time a resource-accessing task is migrated to its processor. When the timer triggers, it sets the task's priority to the corresponding ceiling priority and invokes the scheduler to check whether a higher priority task is ready to execute. If the holder releases the resource during its NP section, the timer is then cancelled.

V. Evaluation

With the proposed facilities implemented, experiments are conducted to (1) gather run-time overheads of the new implementation; (2) demonstrate the impact of the migration issues and (3) verify the effect of proposed solutions. The experiments are performed by the implementations in [20] on a Intel Core^{TM} i7-6700K with a base frequency of 4.0 GHz. During evaluation, hyper-threading on each core is disabled; core 0 is preserved to handle interrupts; core 1, 2, 3 are isolated from the system and the network is disabled.

A. Primitives Overheads

The first experiment is to reveal the run-time overheads of the MrsP implementation, including locking a resource, releasing the lock and migrating a lock holder. In our implementation, obtaining a lock requires the updates of the task priority, FIFO queue and the data structure of the lock within maximum observed time of 150 ns. Lock releasing can be finished within 100 ns, which restores the task priority, updates the FIFO queue and lock structure. If a task is on a remote processor after releasing the lock, it will be migrated back to its original processor by Litmus^{RT} within 2200 ns. In the case where a resource holder is preempted and needs migration, it is placed into the preemption queue and then resumes on a remote processor by pull operation. The whole procedure takes 7000 ns.

B. False Migrations

To investigate the frequency of false migrations, pressure testing is conducted. The testing program contains three resource requesting tasks on each core as well as three high priority tasks with very short periods (500 μs). Table I gives the total number of migrations triggered by the helping mechanism and the number of false migrations occurred in 100,000 jobs. The test is conducted by a MrsP implementation with generic pull and push operations (MrsP-generic) and the new MrsP implementation (MrsP-new). As shown in the table, the generic implementation has a failure rate of 2.14%. In addition, the number of false migrations is theoretically unbounded and can increase with the increase of parallelism and the number of releases of high priority tasks on each core. However, no false migration occurred in the new MrsP implementation and fewer migrations are triggered as no further migrations are needed to recover from the false ones.

TABLE I
False Migrations in 100,000 executions

Implementation	Total Migrations	False Migrations	Failure Rate
MrsP-Generic	598,107	12,813	2.14%
MrsP-New	428,618	0	0%

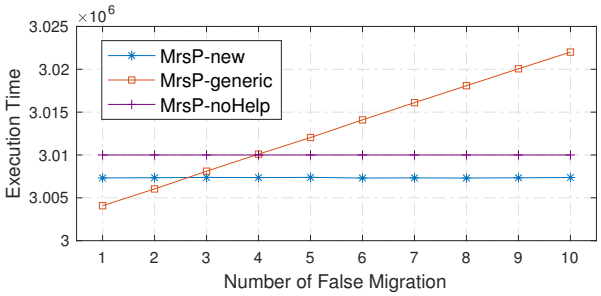

Fig. 3. The impact of false migrations on the critical section execution time

The following experiment demonstrates the impact of false migrations on the execution time. As the false migration is caused by race conditions and is difficult to reproduce on each release, we simulate its affects by preventing the migrated holder from being scheduled. In this test, the length of critical section is 3 ms and the computation time of the preemptor is 10 μs. As shown in Figure 3, the execution time under MrsP-new (3.007 ms) is not affected by false migrations.

As for MrsP-generic, although it has a lower cost for each migration, the execution time is prolonged by false migrations and is higher than that of MrsP-new with more than 2 false migrations. In addition, its execution time exceeds the time with the helping mechanism disabled (MrsP-noHelp) with more than 3 false migrations. Under such situations, MrsP has a poor efficiency and can be outperformed by protocols with a simple ceiling priority facility.

C. Frequent Migrations

To illustrate the frequent migration issue, pressure testing is conducted with a two-core system. On each core, there exist a high priority task with a computation time of 2 ms and a resource requesting task with a critical section length of 1 ms. All tasks will be released again immediately after they finish. With the generic approach where the holder can be preempted anytime, we measured the maximum execution time of 8 ms with an average of 6.1 migrations. Yet by applying the NP section with a length of 14 μs (a doubled migration cost), the holder has a lower execution time of 6.5 ms and 2.3 migrations each time it accesses the resource.

TABLE II
MIGRATIONS AND EXECUTION TIME UNDER MRSP-NP

	Migrations	Execution Time	Standard Deviation
MrsP-Generic	99,807	7.18×10^8 ns	171.76
MrsP-NP	70	1.5×10^6 ns	437.18

To further illustrate the efficiency of the NP section, a test is performed to preempt the resource holder each time after it is scheduled. Upon each preemption, the help mechanism will be triggered and the holder will be pulled to execute on a remote processor. The results are given in Table II. With the original MrsP, the resource holder suffers from frequent migrations, which leads to a huge resource execution time. Yet with the NP approach, the holder can only be preempted after it executes for 14,400 ns (the length of the NP section) so that it only suffers from 70 migrations and has a much lower resource execution time.

VI. CONCLUSION

In this paper, we conducted an investigation towards the correctness and efficiency of implementing MrsP in fully partitioned systems. We identified two major problems due to its migration-based helping mechanism when applied in fully partitioned systems : (1) false migrations and (2) frequent migrations. We demonstrated that each of the issues can cause excess migrations, which impose a huge amount of run-time overheads and greatly undermine the efficiency of the protocol. A false-migration-free facility and NP section are then introduced to prevent such issues and to guarantee the progress of task execution with resources after each migration. Our evaluation results demonstrate that the migration issues are successfully addressed by proposed solutions, which require less migrations when accessing resources and provide an improved performance of the protocol in practice.

REFERENCES

[1] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[2] Aaron Block, Hennadiy Leontyev, Björn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA 2007, 13th IEEE*, pages 47–56. IEEE, 2007.

[3] Björn Brandenburg and James Anderson. A comparison of the M-PCP, D-PCP, and FMLP on litmusrt. In *Proceedings of the 12th international conference on principles of distributed systems*, pages 105–124. Springer, 2008.

[4] Björn B Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.

[5] Björn B Brandenburg and James H Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in litmusˆ rt. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 185–194. IEEE, 2008.

[6] Alan Burns, Marina Gutierrez, Mario Aldea Rivas, and Michael González Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Transactions on Computers*, 64(5):1241–1253, 2015.

[7] Alan Burns and Andy Wellings. A schedulability compatible multiprocessor resource sharing protocol– MrsP. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 282–291. IEEE, 2013.

[8] Sebastiano Catellani, Luca Bonato, Sebastian Huber, and Enrico Mezzetti. Challenges in the implementation of MrsP. In *Reliable Software Technologies–Ada-Europe 2015*, pages 179–195. Springer, 2015.

[9] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[10] Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 90–99. IEEE, 2010.

[11] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 73–83. IEEE, 2001.

[12] Jorge Garrido, Shuai Zhao, Alan Burns, and Andy Wellings. Supporting nested resources in mrsp. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 73–86. Springer, 2017.

[13] Wen-Hung Huang, Maolin Yang, and Jian-Jia Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 111–122. IEEE, 2016.

[14] Ragunathan Rajkumar, Lui Sha, and John P Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, volume 88, pages 259–269, 1988.

[15] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[16] Junjie Shi, Kuan-Hsun Chen, Shuai Zhao, Wen-Hung Huang, Jian-Jia Chen, and Andy Wellings. Implementation and evaluation of multiprocessor resource synchronization protocol (MrsP) on litmus rt. In Operating Systems Platforms for Embedded Real-Time Applications (OSPERT). 2017.

[17] Hiroaki Takada and Ken Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *The 18th IEEE Real-Time Systems Symposium Proceedings*, pages 134–143, 1997.

[18] B Ward and J Anderson. Nested multiprocessor real-time locking with improved blocking. In *Proceedings of the 24th Euromicro conference on real-time systems*, 2012.

[19] Alexander Wieder and Björn B Brandenburg. On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 45–56. IEEE, 2013.

[20] Shuai Zhao. Implementing MrsP on fully partitioned systems. https://github.com/RTSYork/mrsp/blob/master/ImplementatingMrsP.pdf, 2016. accessed by 17/June/2016.

[21] Shuai Zhao, Jorge Garrido, Alan Burns, and Andy Wellings. New schedulability analysis for MrsP. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2017.