# Implementation and Evaluation of Multiprocessor Resource Synchronization Protocol (MrsP) on LITMUS$^{RT}$

Junjie Shi[1], Kuan-Hsun Chen[1], Shuai Zhao[2], Wen-Hung Huang[1], Jian-Jia Chen[1], and Andy Wellings[2]
[1]Department of Informatics, TU Dortmund University, Germany
[2]Department of Computer Science, University of York, United Kingdom

# Implementation and Evaluation of Multiprocessor Resource Synchronization Protocol (MrsP) on LITMUS$^{\text{RT}}$

Junjie Shi[1], Kuan-Hsun Chen[1], Shuai Zhao[2], Wen-Hung Huang[1], Jian-Jia Chen[1], and Andy Wellings[2]
[1]Department of Informatics, TU Dortmund University, Germany
[2]Department of Computer Science, University of York, United Kingdom
{junjie.shi, kuan-hsun.chen, wen-hung.huang, jian-jia.chen}@tu-dortmund.de[1]
{zs673, andy.wellings}@york.ac.uk[2]

***Abstract*—Preventing race conditions or data corruptions for concurrent shared resource accesses of real-time tasks is a challenging problem. By adopting the resource synchronization protocols, such a problem has been studied in the literature, but there are not enough evaluations that consider the overhead from the implementations of different protocols. In this paper, we discuss our implementation of the Multiprocessor Resource Sharing Protocol (MrsP) and the Distributed Non-Preemptive Protocol (DNPP) on LITMUS$^{\text{RT}}$. Both of them are released in open source under GNU General Public License (GPL2). To study the impact of the implementation overhead, we deploy different synchronization scenarios with generated task sets and measure the performance with respect to the worst-case response time. The results illustrate that generally the implementation overhead is acceptable, whereas some unexpected system overhead may happen under distributed synchronization protocols on LITMUS$^{\text{RT}}$.**

## I. Introduction

When concurrent real-time tasks have to access shared resources, ensuring the timeliness is a challenging problem. To prevent race conditions or data corruptions, concurrently accessing the same resource is prohibited by exploiting mutual exclusion. That is, when a task has already been granted to access a shared resource, any other tasks cannot access the shared resource at the same time. To realize mutual exclusion in operating systems, *semaphores* are widely used. However, using semaphores introduces other problems, e.g., deadlocks and priority inversions. Towards this, many resource synchronization protocols have been proposed to prevent such problems caused by such shared (logical) resources.

In uniprocessor systems, the Priority Ceiling Protocol (PCP) has been widely accepted and supported in real-time operating systems. Nowadays, multiprocessor platforms have been widely used. There have been several resource synchronization protocols proposed for multiprocessors. Specifically, in a recent paper by Huang et al. [9], four sound protocols, i.e., Multiprocessor Priority Ceiling Protocol (MPCP) [11], Distributed Priority Ceiling Protocol (DPCP) [12], Distributed Non-Preemptive Protocol (DNPP) [9], and Multiprocessor Resource Sharing Protocol (MrsP) [6], have been discussed.

To schedule real-time tasks on a multiprocessor platform, there are mainly three classes of scheduling algorithms: *global*, *partitioned* and *semi-partitioned* algorithms. Brandenburg et al. [5] have recently shown that global scheduling is probably not necessary for scheduling independent and implicit-deadline sporadic (or periodic) tasks. Even though there already exist several resource sharing protocols, there is no clear comparison among those protocols. One open problem is the proper task partitioning algorithms suitable for different multiprocessor resource synchronization protocols. The resource-oriented partitioned fixed priority (P-FP) scheduling proposed by Huang et al. [9] is pragmatically good for the DNPP and DPCP. There are task partitioning algorithms for the MPCP proposed in [10], but they are in general dominated by the resource-oriented partitioned scheduling from [9]. As for the MrsP, to the best of our knowledge, there is no specific discussion for task partitioning yet.

The evaluations in the literature are still mostly based on the theoretical analyses without considering the overhead introduced by the real-world implementation. Due to the fact that the induced run-time overhead is not negligible in some protocols, a theoretically good protocol may perform worse than other protocols that only need low overhead in the real world implementation. Although Brandenburg et al. [4] already considered the runtime overhead on the LITMUS$^{\text{RT}}$ into schedulability tests statically, they did not consider the MrsP and the impact of the task partitioning.

Among the above four protocols, the MrsP was proposed in 2013 by Burns and Wellings [6]. It strikes a compromise between short and long shared resource accesses. Specifically, the MrsP has two important features: 1) it uses spin-lock to handle short resources accesses preferably, and 2) it has a helping mechanism to reduce the indirect blocking from the long resources accesses. The helping mechanism enables a task waiting to gain access to a resource to undertake the associated computation on behalf of any other waiting tasks. However, the helping mechanism makes the MrsP not easy to be implemented and verified. To the best of our knowledge, before this paper, there was only one implementation published by Catellani et al. [7]. Their implementation has been included as a stable version in Real-Time Executive for Multiprocessor Systems with the latest version 4.11 [1]. Catellani et al. [7] also presented their implementation on LITMUS$^{\text{RT}}$. However, their implementation on LITMUS$^{RT}$ is not publicly available.[1]

To have a comprehensive performance evaluation on the same basis as [4], in this paper, we still consider the incurred overhead on LITMUS$^{RT}$. Based on LITMUS$^{RT}$, we release our implementation as a patch of the MrsP supporting global multi-resources non-nested accesses, and discuss the difficulties and the potential pitfalls during the implementation process in detail. To evaluate the performance of each protocol with respect to the worst-case response time, different resource synchronization scenarios are studied under different protocols.

**Our contributions:**

- The difficulties of the implementation are discussed. We release an executable version of the MrsP in [13] supporting non-nested multi-resources sharing on LITMUS$^{RT}$. The DNPP is also released by extending the original DPCP implementation on LITMUS$^{RT}$.
- We evaluate the real-world overhead of each routine in synchronization protocols on LITMUS$^{RT}$, i.e., migration, context switch, and helping mechanism.
- The performance of four protocols (*i.e.,* the MPCP, the DPCP, the DNPP and the MrsP) are evaluated with respect to the measured worst-case response time. Some interesting case studies are shown to illustrate different suitable scenarios of resource sharing among all the considered protocols.

## II. Multiprocessor resource sharing Protocol

In this section, we introduce the concepts of the MrsP briefly and discuss the difficulties of the implementation. MrsP was developed by Burns and Wellings in [6] and has the following properties:

- All available resources are assigned to a set of ceiling priorities. Each resource has one ceiling priority per processor depending upon the priorities of tasks which use it. For processor $p_k$, the ceiling priority of a resource is the maximum priority of all tasks allocated to processor $p_k$ using that resource.
- For any resource, the priority of the task which requests that resource is immediately raised to the local ceiling of that resource.
- The sequence of accessing to a resource is handled in a FIFO order.
- Every task waiting to gain an access of a resource must be capable of undertaking the associated computation on behalf of any other waiting task. Any cooperating task must undertake the outstanding requests in the original FIFO order. In the rest of the paper, we follow [7] to call this property as *the helping mechanism*.

Overall, the interplay within the protocol among each component is shown in Figure 1. From the right-hand side of Figure 1, once a task requests a resource, it spins by

[1]Quoting the message from Prof. Enrico Mezzetti, "... the implementation was based on the 2013.1 (now deprecated) version of LITMUS-RT." Although we received the courtesy source code from them, we are not able to execute the protocol.

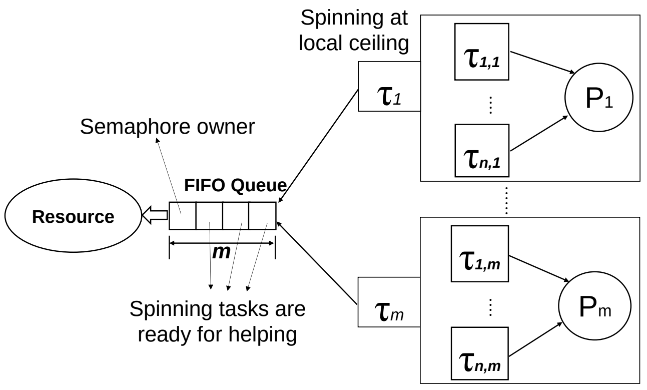

Fig. 1: Overview of resource sharing scenarios under the MrsP

setting its priority level to the local ceiling priority in that processor immediately. Such an operation can guarantee that there is only one task in requesting the same resource from that processor. In other words, this ensures that the maximum number of jobs in the waiting queue of a resource is at most the number of processors in the system. The waiting queue of the corresponding resource is managed in FIFO order. Moreover, all the tasks which are spinning on their processors are ready to help the semaphore owner. Details of the implementation are illustrated in the following subsections.

### A. FIFO Spin Lock

We apply the *ticket-based* spin lock [14] in our implementation, which is a spin lock and can guarantee the FIFO order for the requests of a shared resource. It consists of two components, a ticket variable and a grant variable. Arriving threads atomically *fetch-and-increment* the ticket and then spin, waiting for the grant variable to match the value returned by the *fetch-and-increment* primitive. At that point the thread owns the lock and may safely enter the critical section. The pseudo code example can be found in [14].

### B. Spinning at Local Ceiling

Each resource on each processor has its local ceiling, defined as the highest priority among all the tasks (on that processor) that request that resource. The boosting of the priority of the spinning task to the local ceiling can ensure that there is at most one task on that processor requesting the same resource, since the other tasks requesting the same resource will be en-queued into the ready queue due to the definition of the local ceiling priority. In our implementation, the local ceiling priority is calculated by users and given to the system statically. When one task finishes its critical section, the priority will be lowered to the original one. In order to ensure that the task can return back to the original status, the processor and the priority information are saved in advance before it can enter to its critical section.

### C. Helping Mechanism

The helping mechanism allows a spinning task on a processor to help other tasks on other processors. Since spinning wastes the available computation power, when it is possible, helping other preempted tasks can improve the performance of the system. In the MrsP, the tasks that can be *helped* are the ones that are preempted but already own semaphores and have entered their critical sections. A task is a *semaphore owner* if the semaphore is currently locked by the task. In the original design by Burns and Wellings, several rules were introduced in Section VI in [6] to implement the helping mechanism of MrsP. Furthermore, Catellani et al. explained in Section 3.3 in [7] why implementing the helping mechanism is a challenging task. Prior to this work, the helping mechanism in MrsP was introduced by using itemized rules. In our view, these rules can be summarized by two scenarios, defined as Pull and Push as follows:

***Push:*** In this case, the preempted semaphore owner can migrate itself *actively* to the processor of the spinning task, which is waiting for the semaphore. To have this situation, there is a task $\tau_s$ spinning on its processor $m_s$ before the moment that semaphore owner $\tau_o$ of semaphore $R$ is preempted by another task on processor $m_o$ where $m_s \neq m_o$. When task $\tau_o$ is preempted by a higher-priority task on processor $m_o$, it is migrated to the helper's processor, i.e., $m_s$ in this case. The $\tau_o$'s priority will be set to a value which is one bit higher than the priority of $\tau_s$, in order to preempt the spinning task $\tau_s$ on processor $m_s$.

To successfully implement this mechanism, we need to identify whether such a task $\tau_s$ exists or not. If there are multiple tasks spinning for being granted to access semaphore $R$, we have to decide one of them to be the helper. To find out the helper for the semaphore owner $\tau_o$, the FIFO order is used. That is, among all the spinning tasks that are waiting for $R$, task $\tau_o$ is helped by the task that is currently spinning on its processor without being preempted by following the FIFO order. Here are the details of our implementations:

- Once the *scheduler()* notices that the semaphore owner is preempted on processor $m_o$, the processor id of the semaphore owner is set to a negative number.
- Then, the function *finish_switch()* will mark the situation and try to find a helper for the semaphore owner.
- A field *current.next* is used to point to the next task which is requesting the same resource. This parameter is set when it is getting the ticket, so that the helpers can be sorted in a FIFO queue. The function *finish_switch()* will traverse the semaphore owner's possible helper list, to find out whether there is a task spinning at its processor and ready for help.

The above explanation is the simplest case without further preemption on a semaphore owner. In fact, a semaphore owner may be preempted while it is helped by other tasks. If so, the semaphore owner can be further helped by other spinning tasks. In our implementation, we only check whether the semaphore owner can migrate back to its original processor (recall that each task under the MrsP is assigned to one processor originally due to task partition) and continue to execute or be helped by other spinning tasks. If the semaphore owner cannot proceed to be executed, it will be en-queued to the corresponding processor's ready queue and the flag *sem_owner_preempted* will be marked as one.

***Pull:*** In this case, the semaphore owner $\tau_o$ of semaphore $R$ has been preempted on processor $m_o$. After a while, one task $\tau_s$ is released and spinning on another processor $m_s$ trying to lock the semaphore. At that moment, the semaphore owner $\tau_o$ was already en-queued to the ready queue on processor $m_o$. Therefore, the semaphore owner has become passive, and the helper has to actively check whether the semaphore owner is still executing or is already preempted. Once task $\tau_s$ finds that the flag *sem_owner_preempted* is set to one, which indicates that the semaphore owner has been already preempted, the helper will get the run-queue lock from the processor where the semaphore owner is located, and help the semaphore owner $\tau_o$ to migrate to the processor $m_s$.

Similarly, we also need to consider the situation if the *next* semaphore owner is preempted during its spinning time. Once the *scheduler()* notices the spinning task is preempted, the parameter *preempted_while_waiting* will be set to one. When the last semaphore owner releases the resource and the next task is noticed to have been preempted while waiting, the parameter *sem_owner_preempted* will be set to one, so that the potential helper can make a help.

## *D. Implementation Overhead and Potential Deadlock*

To achieve the aforementioned two techniques, i.e., ticket-based spin lock and helping mechanism, we added several elements into the *rt_params* structure which is originally used to define the property for each task, i.e., priority, period, execution time, etc. In the *rt_params* structure, the keyword *volatile* was adopted on the ticket mechanism as well as other variables which may be updated frequently. The usage of *volatile* keyword can avoid the optimization on subsequent reads or writes in compilation phase; otherwise, potential errors like incorrectly reusing a stale value or omitting writes may take place. To implement the semaphores under the MrsP, we created a new structure named *mrsp_semaphore*, in which the operation *atomic_t* supported by the standard Linux kernel is applied to define the variables which may be read or written concurrently, i.e., *serving_ticket*, *sem_owner_preempted*. The atomic operation can protect these variables from concurrent accesses. However, using both techniques, i.e., the keyword volatile and atomic operations, may cause significant run-time overhead which also influences the performance of protocols.

Moreover, deadlock had occurred in our early implementation when we followed the standard usage of ceiling protocols, in which the local resource ceiling was set as the highest priority of the task which requests the resource on that processor. However, in the current scheduling strategy on LITMUS$^{\text{RT}}$, when two tasks have the same given priorities, the task with the lower PID number has the precedence in the system on LITMUS$^{\text{RT}}$. This feature results in potential deadlocks, which is illustrated in Figure 2.

Task $\tau_2$ is released at $t_0$ and starts its normal execution. It enters its critical section at $t_1$, and the priority is raised to the resource ceiling. Task $\tau_1$ is released at $t_2$, and it can

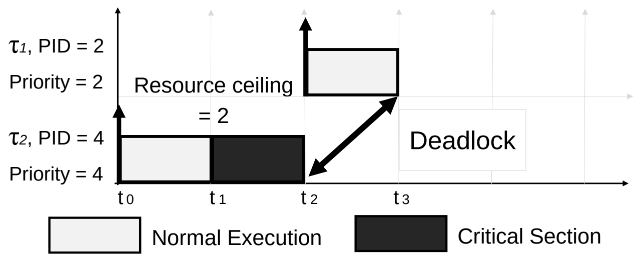

Fig. 2: Deadlock possible for the MrsP on LITMUS$^{RT}$

preempt $\tau_2$ even when they have the same priority under the current scheduling strategy on LITMUS$^{RT}$. At $t_3$, $\tau_1$ requests the resource which is held by $\tau_2$, but $\tau_2$ is already preempted by $\tau_1$. Thus, deadlock occurs. To prevent this situation, we have two choices: 1) set the ceiling priority a bit higher than the real one; 2) change the rule of the judgment of scheduling strategy for two tasks with same given priorities. If two tasks have the same priority, the first coming task has the higher priority. In our implementation, we use the second option. By this modification, once a task starts the execution of its critical section, it cannot be preempted by any tasks which may request this resource in this processor. Therefore, no deadlock will happen.

## III. Overhead Comparison and Discussion

To evaluate the overhead of our implementation, we follow the latest work from Huang et al. [9] to compare four sound protocols, i.e., the MPCP, the DPCP, the DNPP, and the MrsP, with the real implementation on LITMUS$^{RT}$. Since all the implementations are based on the plug-in *partitioned fixed-priority* (P-FP) on LITMUS$^{RT}$, the overhead for some common routines are the same, e.g., migration and context-switch. The measured overheads on LITMUS$^{RT}$ are shown in Table I (see Section IV for detailed setup). Except MPCP, other three protocols suffer from the migration overhead. DNPP reduces the context switch overhead comparing to DPCP. In Table I, the help overhead is the additional effort needed by the scheduler to support the helping mechanism when making scheduling decisions under the MrsP (migration overhead has not counted). Naturally, the MrsP also has the migration overheads due to the helping mechanism.

| Routine | Migration | Context Switch | Help |
|---|---|---|---|
| Avg. Time | 5.6 $\mu$s | 1.5 $\mu$s | <1 $\mu$s |

TABLE I: Routine overheads among different protocols

Besides the implementation of the MrsP, which is discussed in the previous session, the MPCP and the DPCP we used are originally supported on LITMUS$^{RT}$. However the DNPP is not supported yet. To realize the DNPP, we add a pair of non-preemptive flags named *np_enter* and *np_exit* for the critical section in the user space, since the scheduler on the LITMUS$^{RT}$ kernel supports non-preemptive executions if these non-preemptive flags have been set to 1's. Similar to the *non-preemptive protocols* in uniprocessor system, once a task in a critical section has started to be executed, it cannot be preempted until it finishes under the DNPP. Although the overhead of context switch is greatly reduced by using non-preemption, the blocking time for each task may cause higher blocking time than using the DPCP. For instance, when one task with a lower priority enters to a long critical section by which other high priority tasks cannot access shared resources. In the DPCP, the maximum blocking time of a task is dominated by the longest critical section among tasks which require the same resource with lower priorities. Nevertheless, the maximum blocking time of tasks under the DNPP is decided by the longest critical section among other tasks even without using the same resource in the same synchronization processor.

Due to the different ways of handling waiting tasks, these four protocols can be distinguished into two classes: *suspension-based* and *spin-based*. Under *suspension-based* protocols, e.g., the MPCP[2], the DPCP, and the DNPP, tasks waiting for a global resource suspend and are en-queued in an associated prioritized global wait queue. A task blocked by a global resource suspends and makes the processor available for the local tasks. Under the spin-based protocol, the task blocked by a global resource spins on that processor unless there is another higher priority task coming. As a *spin-based* protocol, MrsP has advantages on short resources accessing with less context switch overhead; suspension-based protocols have advantages on long resources accessing with full usage of processor capability. For the fairness, we prepare these two scenarios of resource usages as our case studies in Section V, i.e., short and long resource accesses, to evaluate the benefits of using different protocols on LITMUS$^{RT}$.

## IV. Experimental Setup

The hardware platform used in our experiments is a cache-coherent SMP consisting of four 64-bit Intel i7-5600U processors running at 2.6 GHz, with 32k L1 instruction caches as well as 32k L1 data caches, a 256k L2 cache, 4096k L3 caches and 8 GB of main memory. We adopt the build-in tracing toolkit to measure the overheads and collect the performance data, which is an efficient low-overheads toolkit proved in [2].

### A. Task Set Choosing

In this paper, we generated 7 kinds of periods and 40 tasks in total. We defined the utilization for each task between 0.1% and 10%. Due to the limitation of the build-in tracing toolkit, we arrange the number of tasks with different periods for an acceptable experiment duration by following the normal distribution as shown in Table II. The priorities of tasks are assigned under Rate-Monotonic scheduling, i.e., the shorter the period is the higher the priority is. Once two tasks have the same period, the task with the higher utilization has the higher priority.

[2]Suspension-based and spin-based MPCP are both supported on LITMUS$^{RT}$. In this paper, we adopt the original suspension-based MPCP [11].

| Period (ms) | 5 | 10 | 20 | 50 | 100 | 200 | 1000 |
|---|---|---|---|---|---|---|---|
| # of tasks | 2 | 5 | 8 | 10 | 8 | 5 | 2 |

TABLE II: The number of tasks with different periods

From the evaluation of [9], we can find that when the total utilization varies from 120% to 280% for 4-processor system, the performance for each protocol shifts rapidly. Thus, in our experiment, we consider 5 different total utilization from 120% to 280% and each step is 40%. In order to meet all the aforementioned constraints, i.e., the total number of tasks, the utilization for each task, total utilization, we adopted arithmetic progression to generate the utilization for each task in which a task with a higher priority has higher utilization. For each task, the expected execution time emulated by the *rtspin* tool is equal to the utilization multiplying the period for each task. For each task, we adopted normal distribution to vary the expected execution time for each task to emulate the various execution time of jobs in the reality. We set the average case execution time equals to 90% of the WCET and best case execution time as 50% of the WCET. If the generated execution time is out of the range between 50% and 100% WCET, it will be set as the boundary value according to which boundary it is close to.

## B. Shared Resources Allocation

As discussed in Section III, different protocols have their advantages on different resource synchronization scenarios. For the fairness, we define two types of resource accesses with the constant ranges: short resource access $0\mu s < R_{short} \leqslant 100\mu s$; long resource access $200\mu s < R_{long} \leqslant 300\mu s$. We set the possible lengths of generated resource execution time as a range of constant numbers rather than the percentage of the execution times. If we set the resource accessing time using percentage scaling, for those tasks with long generated execution time, the execution time for the critical section will be very large, which makes the system difficult to be scheduled by any of the studied synchronization protocols.

However, if the generated resource access length is larger than the execution time of one job, we still have to use the percentage scaling: under single resource accessing, $R_{short} = 20\% \times$execution time, $R_{long} = 80\% \times$execution time; under multi-resources accessing, $R_{short} = 20\% \times$execution time, $R_{long} = 30\% \times$execution time. To the end, we choose the following four resource access scenarios in the evaluation:

- **R1_short:** only one short resource is available and each task requests it at most once.
- **R1_long:** only one long resource is available and each task requests it at most once.
- **Multi_short:** six short resources are available and each task requests three of them.
- **Multi_long:** six long resources are available and each task requests three of them.

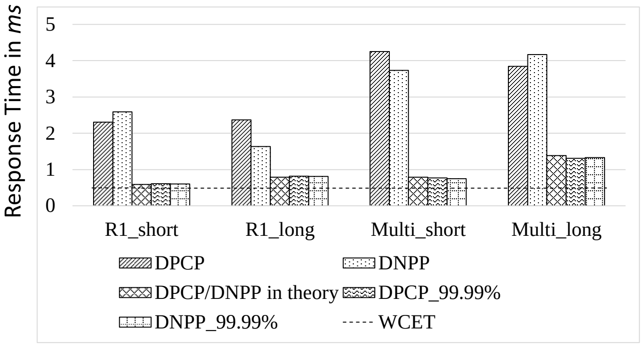

Fig. 3: Analysis of the WCRT with total utilization 200%

## C. Partition Algorithm

Since the four protocols considered in this paper are all based on partitioned or semi-partitioned scheduling, the partition algorithm could play an important role. Since we are not aware of any efficient partition algorithm for all synchronization protocols, we follow [9] and adopt its proposed heuristic partition algorithm for the DPCP and the DNPP. For the MPCP and the MrsP, the partition algorithm we used can be described as follows: (*1*) sort all the tasks by their priorities; (*2*) calculate the utilization for each processor, e.g., the total utilization is 200% and we have four processor, which implies that the utilization for each processor is 50% if we can perfectly partition the tasks; (*3*) allocate the tasks to the processors, starting from the highest-priority task to the first processor until the utilization of that processor reaches to the value that we calculated on step 2, then allocate the next task to the next processor. Please note that, if there is no such processors that can hold the next task, then we assign it onto the processor which has the lowest utilization.

## V. Results and Discussions

In this section, we conducted extensive experiments using task sets generated in Section IV. We had results of 20 groups for each protocol under different shared resources assumptions and utilization settings, where all of them are feasible without deadline misses. To evaluate the performance, we measured the worst-case response time (WCRT) of the highest-priority tasks during the experiments among different configurations. In the real experiments, we expect that the WCRT should be less than or equal to the summation of theoretical value and run-time overheads. Under our configurations, the theoretical value of WCRT of the highest-priority tasks under the DPCP and the DNPP with the same experiment setting are the same.

Since the overall run-time overheads of the DPCP under P-FP plugin on LITMUS$^{RT}$ have been proved to be fairly small in [3, 8], the expected WCRT in the real experiments should be a little bit larger than the value in theory. However, as shown in Figure 3, we found that both the DPCP and the DNPP had unexpected overheads so that the response time was much larger than the theoretical value even running on another AMD platform. Due to unknown system interference, we could not repeat the unexpected overhead with the same settings in every round, so that we were not able to eliminate the unexpected system overhead. Furthermore, we can find out that those jobs with unexpected response times are really rare, i.e., the possibilities are less than 0.01%. By filtering them, the WCRTs over all the other 99.9% jobs under the DPCP and the DNPP are still close to the theoretical values. In the following evaluations, we applied those filtered values under the DPCP and the DNPP for a more sensible comparison.

Figure 4 shows the performance evaluation under the four protocols in terms of the WCRTs of the highest-priority tasks. Intuitively, we can see that the performances of tasks have not varied significantly under different utilization. Under the DPCP and the DNPP, the tasks with the higher priorities may only be blocked once for requesting one resource once; under the MPCP, the most blocking time comes from the executions for critical sections of lower tasks in the same processor; under MrsP, it can be blocked at most four times for requesting one resource once. With the same WCET settings in single resource access scenarios, the tasks with the highest-priorities under the DPCP and the DNPP indeed have the lowest WCRT comparing to the other protocols. However, under multi-resources assumptions, the results under the DPCP and the DNPP cannot always outperform the others under the MPCP and the MrsP, since the additional overhead of the DPCP and the DNPP plays an important role under multi-short resources accessing situation.

## VI. Conclusion

This paper provides the publicly available implementation of the MrsP and the DNPP on LITMUS$^{RT}$, which is available on [13]. Throughout this paper, we can notice that the induced run-time overhead of synchronization protocols is not negligible but acceptable. However it is hard to come out the conclusion which protocol has to be preferred for any specific configuration in the limited spectrum of this study. We hope that this work may encourage more discussions in the future.

## Acknowledgments

This paper is supported by German Research Foundation (DFG), as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/). The authors thank anonymous reviewers for their suggestions on improving this paper.

## References

[1] RTEMS: Real-Time Executive for Multiprocessor Systems. http://www.rtems.com/, 2013.

[2] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, 2007.

[3] B. B. Brandenburg and J. 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. RTCSA 2008. 14th IEEE International Conference on*, pages 185–194. IEEE.

[4] B. B. Brandenburg and J. H. Anderson. A comparison of the m-pcp, d-pcp, and fmlp on litmusrt. In *International Conference on Principles of Distributed Systems*, pages 105–124. Springer, 2008.

[5] B. B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 99–110. IEEE.

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

[7] S. Catellani, L. Bonato, S. Huber, and E. Mezzetti. Challenges in the implementation of mrsp. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 179–195. Springer, 2015.

[8] F. Cerqueira and B. Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2013.

[9] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS), 2016 IEEE*.

[10] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium, RTSS 2009. 30th IEEE*, pages 469–478. IEEE.

[11] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123. IEEE.

[12] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269. IEEE.

[13] J. Shi, S. Zhao, and K.-H. Chen. MrsP-LITMUS-RT. https://github.com/kuanhsunchen/MrsP-LITMUS-RT/, 2017.

[14] Y. Solihin. *Fundamentals of parallel computer architecture*. Solihin Publishing and Consulting LLC, 2009.

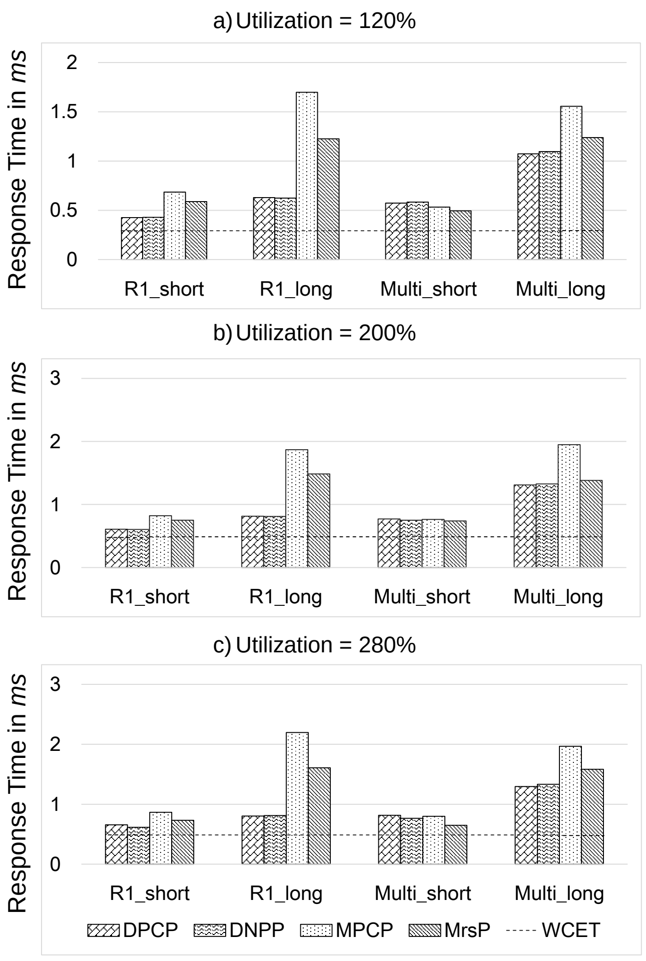

Fig. 4: Worst case response time of the highest-priority task under different total utilizations