Challenges in the Implementation of MrsP

Sebastiano Catellani[1], Luca Bonato[1](✉), Sebastian Huber[2], and Enrico Mezzetti[1]

[1] Department of Mathematics, University of Padua, Padua, Italy
{scatella,lbonato,emezzett}@math.unipd.it
[2] Embedded Brains GmbH, Puchheim, Germany
sebastian.huber@embedded-brains.de

Abstract. The transition to multicore systems that has started to take place over the last few years, has revived the interest in the synchronization protocols for sharing logical resources. In fact, consolidated solutions for single processor systems are not immediately applicable to multiprocessor platforms and new paradigms and solutions have to be devised. The Multiprocessor resource sharing Protocol (MrsP) is a particularly elegant approach for partitioned systems, which allows sharing global logical resources among tasks assigned to distinct scheduling partitions. Notably, MrsP enjoys two desirable theoretical properties: optimality and compliance to well-known uniprocessor response time analysis. A coarse-grained experimental evaluation of the MrsP protocol on a general-purpose operating system has been already presented by its original authors. No clear evidence, however, has been provided to date as to its viability and effectiveness for industrial-size real-time operating systems. In this paper we bridge this gap, focusing on the challenges posed by the implementation of MrsP on top of two representative real-time operating systems, RTEMS and LITMUS^{RT}. In doing so, we provide a useful insight on implementation-specific issues and offer evidence that the protocol can be effectively implemented on top of standard real-time operating system support while incurring acceptable overhead.

Keywords: Real-time systems · Multiprocessor systems · Resource sharing protocols · Empirical evaluation

1 Introduction

Cost, performance and availability considerations increasingly push application developers towards the adoption of multiprocessor platforms even in the traditionally conservative domains of embedded real-time systems [14], [1]. The migration to multicores, however, threatens to disrupt all the analysis approaches and solutions that are consolidated practices on single processors. Despite the notable progress achieved in this direction in recent years [13], scheduling algorithms and schedulability analyses of multiprocessor systems have not reached the same degree of maturity as single processors yet.

In fact, the transition from uniprocessor to multiprocessor continues unabated in spite of the arguably insufficient expertise to master the latter targets in numerous application domains. Under this scenario, *partitioned approaches* to multiprocessor scheduling offer a gentler slope by allowing the user to break down the problem into smaller uniprocessor sub-problems, on which standard consolidated techniques can still be applied. A known drawback of partitioned approaches is that they must undergo the so called *partitioning phase*: an initial step where the system load is broken down in small units, each fitting into a single processor. Partitioning is an NP-hard problem in the general case [18] and has been proved to waste, in the worst case, half of the platform's processing power [3]. The partitioning of a system is not only a problem of sharing processor resources: it should not disregard the implicit constraints stemming from logical resource sharing throughout the system. Prioritizing on system feasibility may enforce a partitioning where two or more logically-dependent tasks are assigned to different partitions, which complicates inter-task interactions.

The conflicting requirements of feasibility (grouping tasks based on a quantitative value – typically their utilization) and program logic (clustering tasks based on their actual collaborative patterns) can be accommodated by adopting a resource sharing protocol. Also in this respect, however, state-of-the-art uniprocessor resource sharing protocols cannot be directly applied as they cannot handle resources shared by tasks allocated to different partitions. A specific global resource sharing protocol for multiprocessor systems must be used.

Several global resource sharing protocols have been proposed in the literature. Although capable of guaranteeing mutually exclusive accesses to global resources, not all the proposed solutions are fully satisfactory with respect to the induced costs, both as theoretical and runtime overhead. The induced costs vary enormously between uniprocessor and multiprocessor protocols. In the uniprocessor case, when using an optimal resource sharing protocol, such as the Stack Resource Protocol (SRP) [4], the theoretical overhead stems from the priority inversion suffered by tasks, which is bounded by the length of the critical section. In the multiprocessor case, instead, the simple fact that a resource can be contended for in parallel (and not simply concurrently) intrinsically amplifies the effects of priority inversion. Moreover, in partitioned systems, it is necessary to determine a criterion to assign urgency of tasks using or waiting for global resources on different processors: we may want to reduce as much as possible the time a remote task waits for an already locked resource while delaying other tasks not interested in that resource.

These same concepts are recalled by the principle of optimality for resource sharing protocols for global [9] and partitioned systems [11],[7]. When it comes to partitioned systems, optimal solutions exploit ordered queues to avoid starvation while serializing the access to shared resources. A *helping mechanism* is advocated to speed up the release of a resource without hindering unrelated tasks. This mechanism can consist in permitting migration of tasks across partitions[1]: when a task holding a resource is not executing, it migrates to a partition

[1] Migration here is determined by the protocol and not due to a general scheduling decision, as in globally scheduled systems.

where there is a task waiting for the same resource and that has the possibility to execute but cannot progress until the task relinquishes the resource.

Contributions. Sometimes elegant theoretical solutions show unexpected drawbacks when evaluated against a realistic implementation as they may unveil viability issues and exhibit untenable runtime overheads. In this paper we focus on the Multiprocessor resource sharing Protocol (MrsP) [11], an optimal multiprocessor resource sharing protocol which explicitly targets partitioned systems. Our goal is to gather evidence that such a protocol can be efficiently implemented in standard RTOS. Specifically, in this work we try to point out difficulties and problems hidden behind the theoretical definition of the protocol and that must be addressed for a reference implementation of MrsP. As a complementary objective, we aim at assessing the protocol with respect to the incurred runtime overhead, so as to understand the induced costs (as compared to not having MrsP) and to obtain sound figures to feed into schedulability tests.

The remainder of this paper is organized as follows: in Section 2 we briefly introduce MrsP and the real-time operating systems (RTOS) on which we implemented and evaluated the effectiveness of the protocol. In Section 3 we point out the main challenges and issues encountered in the implementation of MrsP: we discuss possible design choices and detail on the specific solutions adopted on each RTOS. In Section 4 we assess the runtime overheads and performances of our implementations. Section 5 discusses relevant related works. Finally, in Section 6 we summarize our efforts and outline our future lines of work.

2 Background on MrsP and Target RTOSes

Before delving into the subject matter, we provide first a high-level description of the MrsP protocol and briefly introduce LITMUS^{RT} and RTEMS, the RTOS we used as targets in our implementation and evaluation.

MrsP. MrsP [11] is an optimal resource sharing protocol for multiprocessor systems, explicitly developed with the intent of being fully compatible (analyzable) with the standard response time analysis (RTA) framework, similarly to SRP [4] in single processor systems. The timing effects of the protocol can be simply fed into the RTA iterative equation as an additive factor, whose order of magnitude is proportional to the potential parallel contention incurred by global resources.

The core concept of MrsP is closely inspired by SRP: a resource request triggers a change of priority for the requesting task (to the local ceiling) and the task busy waits until it is granted the resource. Requests are organized (and satisfied) according to a FIFO ordering, as represented in Figure 1. A ceiling is defined within each processor, corresponding to the highest priority among all the tasks that may require access to a resource within the partition. This preserves independence of all tasks with priority higher than the ceiling. In order to contain the waiting time, MrsP provides a helping mechanism to enable a task waiting for a resource already locked by a preempted task on a different partition, to allow the lock holder to progress within its critical section: this way the resource can be relinquished faster and used by the next-in-order task. In [11], two solutions are suggested: (i) if the critical section is stateless, then it is sufficient that one waiting task executes in place of the actual resource holder; (ii) in the more general case, with stateful resources, the resource holder should be allowed to migrate to and then to execute in the partition where there is a task busy-waiting for the same resource.

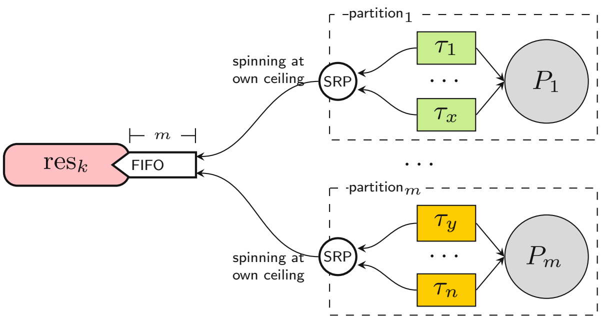


Fig. 1. Visual representation of MrsP

From a theoretical point of view, the contribution of SRP to the RTA equation for a fixed-priority scheduler has been given in [11]. The MrsP algorithm allows each partition to be regarded as a plain uniprocessor system, with the sole exception that the RTA equations need to be updated to account for parallel contention on global resources:

$$R_i = C_i + B_i + \sum_{\tau_j \in lhp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1}$$

where the response time R_i for task τ_i is defined as the sum of three terms. The execution time contribution of τ_i itself (first term), the maximum blocking time induced by SRP, that is determined by the maximum computation time of all resources that are shared between a lower priority task and a task with priority greater or equal to τ_i (second term) and the interference caused by *local* higher priority tasks ($lhp(i)$ in the third term). The worst-case execution time C_i of τ_i can be broken down to explicitly represent the time spent to execute within each resource and the effect of serialization:

$$C_i = WCET_i + \sum_{r^j \in F(\tau_i)} n_i e^j \tag{2}$$

where n_i represents the number of times τ_i uses r_j (in the set of accessed resources $F(\tau_i)$), and e^j represents the effect of serialization (the cost of executing within the resource for each processor whose tasks may access the global resource). As shown in equation 1, a task can be delayed by lower priority tasks for a total duration of B_i. Since this quantity is bounded by the number of processors m, MrsP can therefore be considered an optimal resource sharing protocol [9] since the maximum blocking incurred by a task is $O(m)$.

LITMUS$^{\mathbf{RT}}$. The LInux Testbed for MUltiprocessor Scheduling in Real-Time system [12], [20], abbreviated as LITMUS$^{\mathrm{RT}}$, is a real-time extension of the Linux kernel, aiming at providing a configurable testbed for the implementation of multiprocessor real-time scheduling policies and locking protocols. The LITMUS$^{\mathrm{RT}}$ framework adds an abstraction layer to the execution domain, which provides a set of functionalities and data structures that are compatible with the underlying Linux kernel, regardless of the specific version.

From a scheduling point of view, Linux relies on a list of classes of processes with increasing priority. A specific scheduling policy is associated to each class and a process can execute only if there is no ready process in the higher priority classes. On top of this framework, LITMUS$^{\mathrm{RT}}$ adds a scheduling class at the top of the scheduling hierarchy – thus characterized by the highest priority – and provides an interface to implement the scheduling logic. Through such interface, it is possible to define a specific behavior for each scheduling event (dispatch, job release, blocking, etc.) via a set of primitives. LITMUS$^{\mathrm{RT}}$ also provides a generic interface for implementing locking protocols (based on primitives as lock, unlock, etc.). The system overrides the primitives of a class with those provided by the given implementation.

RTEMS. The Real-Time Executive for Multiprocessor Systems (RTEMS) [23] is an open-source fully featured RTOS that supports a variety of open standard application programming interfaces (API) and interface standards such as POSIX and BSD sockets. In this work we specifically focus on the symmetric multiprocessor (SMP) support in RTEMS, in particular on its scheduling framework. The SMP scheduling framework is structured as a plugin: a set of operations must be provided (e.g., yield, change priority) that are called from within the implementation of the API (e.g, start task). This plugin-like structure makes it possible to assign different schedulers to distinct processors (or partitions).

A fundamental entity that is replicated on each scheduler is the *scheduler node*. Scheduler nodes are used to build up the sets of scheduled and ready tasks and are used to guide the scheduling decisions: each scheduler node corresponds to a specific task and maintains the task's priority with respect to the scheduler. For this last reason a scheduler node is local to a specific scheduler instance and therefore cannot migrate. A scheduler node can be viewed as a box containing a task, a box residing on a specific shelf (a scheduler instance) with other boxes. If a task must migrate then the content of the box is moved from one box to a box of another scheduler instance.

3 Implementation Issues

As summarized in Section 2, MrsP offers an optimal solution for logical resource sharing in partitioned systems that is fully compatible with the standard uniprocessor response time analysis framework. From a practical standpoint, the desirable properties offered by MrsP builds on the provision of three items: (1) a FIFO ordering of global requests; (2) a busy wait mechanism at ceiling priority (until a task reaches the head of the FIFO queue); and (3) a helping mechanism, to speed up the fulfillment of global requests. These three mechanisms together guarantee a nice bound on the effects of serialization. With respect to their actual implementation, some high-level considerations are provided in [11]. However, the focus in [11] is set on the theoretical traits of MrsP and most implementation-specific details were intentionally omitted. Although the three MrsP requirements are relatively simple to accomplish, the way they are actually implemented could largely affect the performance of the protocol.

In the following sections we discuss the main challenges we faced while implementing those three key mechanisms on top of LITMUS$^{\mathrm{RT}}$ and RTEMS, and discuss possible solutions. It is worth noting that among the theoretical helping mechanisms proposed in [11], we decided to stick to the one relying on job migration to support stateful logical resources.

3.1 FIFO Ordering

FIFO ordering for global requests can easily be implemented as an ordered list, either dynamic or static. It is worth noting that MrsP can do with simple fixed-length lists because the maximum length of the FIFO queue is known a-priori, independently from the specific application, as the number of processors available determines the maximum degree of parallelism for the platform. The use of a dynamic list is also a viable option: since each task can participate at most in one FIFO queue, it is possible to statically allocate a node for each task, and then add or remove it from a specific resource list when required.

A more interesting design choice consists in how to organize the FIFO queues in case of nested resources as it may have considerable consequences at run time. With nested resources, as depicted in Figure 2, the helping mechanism will undergo a look-up procedure along several queues to look for an available spinning task to help: clearly, the search space is no more bounded by the length of a single queue. The more intuitive way to combine queues of nested resources is to create a sort of hierarchy (e.g., a tree). The possibility to avoid a deep search is heavily coupled to the busy-waiting technique and the amount of additional information saved in this hierarchy. However, frequent updates to this additional information could end up being even more onerous than sporadic deep searches.

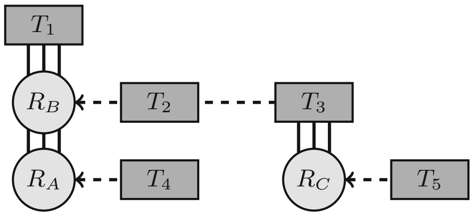


Fig. 2. Scenario with nested resources: the property of being a "task offering help" must be transitive. In this example, task T_5 which is waiting for resource R_C, can help both task T_3, which is directly preventing its execution, *and* task T_1, which is instead doing it indirectly, through R_B. Symmetrically, a helper for T_1 is to be searched within the set of tasks it is directly or indirectly blocking.

RTEMS implementation: the FIFO queues are implemented as dynamic lists where their nodes are created (at system start time) for each task. In case of nested resources these queues naturally form a tree, where all operations performed on resources are bounded by the size of the tree, which could be quite onerous. However, the choice of using the tree finds its motivation in the desire to have a more general structure that could be used to implement several types of semaphores (e.g., using the same structure for both MrsP and OMIP [7]).
LITMUS^{RT} implementation: the FIFO queue is implemented as a list of statically allocated nodes that are dynamically added/removed at need. The current implementation of the protocol does not support nested resources as their implementation would have required a major refactoring of MrsP data structures on LITMUS$^{\mathrm{RT}}$. Anyway, LITMUS$^{\mathrm{RT}}$ provides an optimized implementation of binary heaps that may enable low-latency solutions to this problem.

3.2 Busy Waiting

Busy-waiting is commonly associated to *spinning*, for which several implementations are possible [2], [21]. From our standpoint, the most interesting design decision here is related to whether spinning should be used just to delay the task until it becomes the owner of the resource (i.e., the head of the FIFO queue) or it should be used to check whether the task holding the resource is in need of help (i.e., the resource holder is being preempted and no other task has still offered help). In the first scenario, it is possible to organize the spinning mechanism locally to each task, thus minimizing the interference incurred on shared hardware resources (e.g., bus). In the second scenario, instead, the helping mechanism can be made simpler, at the expenses of a global state for all spinning tasks. Since every spinning task must peek the state of the resource holder, this latter approach could lead to higher contention on hardware resources.

It is worth noting that busy-waiting is not strictly necessary to achieve the intent of MrsP: we only need to prevent all tasks with a priority lower than the ceiling of a partition from resuming execution. This could be achieved, for example, also by suspending all lower priority tasks, or by rising the priority of the idle thread to the ceiling level.
RTEMS implementation: busy wait is implemented as a MCS (Mellor-Crummey and Scott) queue-based locks [21] exploiting the memory hierarchy of the platform: spinning is performed on a local flag, easily fitting inside L1 cache (no bus accesses), whose value is updated only once by the remote task that is releasing the resource (just one bus access).
LITMUS^{RT} implementation: in the baseline implementation a global state is shared among all spinning tasks. A waiting task repeatedly polls the semaphore data structure, which causes high contention on the hardware bus.

3.3 Helping Mechanism

The helping mechanism is surely the most challenging part of the protocol as it is expected to interact with the nominal scheduling operations. The very fact that MrsP introduces job migration into a partitioned system introduces unexpected issues and corner cases.

The strict correlation of the helping mechanism with the scheduling primitives stems from the necessity to enforce the invariant of MrsP, stating that an helping mechanism shall be in place whenever the following conditions hold: (i) a resource holder is not executing; (ii) there is at least one task that is spinning while waiting for the same resource. This invariant must be enforced not only when the resource holder is going to be preempted (and therefore it is necessary to look for a candidate available to help it), but at any scheduling decision. In fact, it may be the case that a spinning task is being resumed while both the resource holder and all other spinning tasks are not executing, and in such case the resumed spinning task must be able to help the resource holder. In practice, this means that the spinning task itself must be able to realize that the holding task is not executing nor already being helped (and then help it) or that a super-partes entity (i.e., the scheduler) recognizes the situation and acts accordingly to enforce the MrsP invariant.

Another problem caused by the helping protocol regards job migrations. A freshly migrated task (the resource holder) is coming from a different partition, with a priority that likely has no meaning in the new partition. Moreover, such task must be able to execute in place of the spinning task (i.e., must be able to preempt it). The solution to this issue is strictly related to the scheduling framework of the RTOS. However, as noted by the authors of MrsP, the issue can be easily solved by updating the priority of the migrated task to a level higher than the priority of the spinning task (which must be equal to the ceiling priority of the resource in that partition). This workaround postulates that the priorities used by the tasks in each partition grows by steps of 2 (so that a migrated task will never hinder a higher priority task of that partition).

A third, possibly subtler problem, is raised by the possibility to migrate tasks while enforcing the ceiling priority in each partition. The definition of MrsP states that a task can execute in a partition different from its own only if it has been preempted in its own partition. This means that whenever the partition of the holding task can execute it, the holding task must be executing there. Therefore, if the holding task is being helped and executes in another partition while in its own partition a local scheduling event makes it available to execute, it should be migrated back. However, this can be less efficient than letting the holding task execute where it migrated to. Even in case the task holding the resource does not migrate back to its own partition, it is still fundamental to preserve the ceiling priority property in that partition: no lower priority tasks should be able to execute. A possible solution to this problem consists in blocking all lower priority tasks until the resource holder completes, which, however, would cause a non-strictly-necessary disturbance to the nominal scheduling operation. A more elegant and efficient way to ensure the ceiling priority property is to let a dummy placeholder execute at the priority of the resource holder. Such placeholder can be created to this end, but the idle thread could be used as well. *RTEMS implementation:* all these issues are managed by the built-in scheduler. The helping mechanism is strictly coupled with the procedures that change the scheduler state. A task that is going to be preempted is moved to another partition by the scheduler to enforce the MrsP invariant. Interestingly, the schedulable entity inside RTEMS are the *scheduler nodes*, which are always updated to point to the task that must be executing (e.g., if the resource holder or the spinning task must execute). Hence, whenever a task is resumed, it is not necessary to check for the MrsP invariant since it is already enforced by the operations that update the scheduler nodes (i.e., obtain and release resource, block, unblock).

Scheduler nodes also simplify the management of priorities of migrated tasks: it is not necessary to change the priority of tasks when they migrate. In fact, the priority of the task with respect to the scheduler is maintained inside a scheduler node, and these nodes do not migrate, only their tasks do. A migrated task takes control of a remote scheduler node and automatically uses the priority of that node (which will be the partition-specific ceiling priority of the resource that the task holder is using since the node belongs to a spinning task, waiting for the same resource). Moreover, scheduler nodes are also used by the idle tasks whenever the rightful owner of a node is executing in another partition: in this way the per-partition ceiling is not violated and the holding task is not forced to migrate back as soon as possible.

LITMUS^{RT} implementation: the invariant of MrsP is enforced in LITMUS$^{\mathrm{RT}}$ by a combination of scheduling and locking primitives. The migration of a preempted resource owner cannot be performed contextually within the context switch: scheduling primitives cannot directly access the list of tasks waiting on a semaphore since both scheduling structures and semaphores are protected through dedicated spin locks, which cannot be simultaneously acquired. Migrations are handled at the end of the scheduling primitive, where the state of the resource holder is checked: if a partition becomes available and the task is not running, a migration will occur. A partition becomes available when a waiting task is resumed or when the processor of the resource owner becomes idle.

These mechanisms are not sufficient to guarantee the MrsP invariant requiring that a (preempted) task holding a resources makes progress whenever at least a task is running and waiting to access the same resource. This is accomplished within the lock and unlock primitives: within the lock primitive, when a task requires the resource and the resource holder is preempted, the protocol triggers a migration; within the unlock, the state of the next resource holder is checked and, if it is not running, the protocol searches an available partition for the migration. This mechanism reduces both the workload of the waiting tasks and the burden of ensuring the invariant of the protocol is charged to the scheduler. In case of migration, the lock holder inherits a priority level above the ceiling of the new partition (destination) in order to preempt the waiting task and to prevent the execution of lower-priority tasks. To this end, each partition uses a specific data structure to keep track of the highest "active" local ceiling, which is equal to the lowest priority available or to the local ceiling of a resource.

4 Evaluation

In this work we focused on the implementation of MrsP on top of two representative RTOSes to assess the protocol in terms of performance and induced overheads. To this end, we performed two groups of experiments on both platforms: on the one hand, we wanted to measure the explicit cost of the primitives involved in the realization of the resource access protocols; on the other hand, we wanted to evaluate the intrusiveness of MrsP, in terms of the additional overhead incurred simply by having the protocol implemented but not used. To complement our evaluation, we conducted a further experiment specific to RTEMS to assess the variation in the cost incurred by the tree-shaped structure used to represent nested resources, with the increase of the nesting depth. Our experiments were conducted on different platforms and execution stacks, according to the support offered by the respective RTOS. For LITMUS^{RT}, experiments were performed on an Intel Quad Core i7-2670QM, running at 2.2GHz. Each core is equipped with a 64KB L1 and 256KB L2 caches, and all four share a 6MB L3 one. The platform includes an internal bus (100MHz) and a serial bus, 5GT/s. The system execution is supported by a Kernel-based Virtual Machine (KVM), providing direct access to the hardware platform without a virtualization middleware, on top of which the LITMUS^{RT} environment is executed (4 physical processors and 512MB of RAM). For RTEMS, the experiments were performed on a Freescale T4240, a 24-processor PowerPC system with 32KB L1 caches and 2MB L2 caches (where 8 processors share one L2 cache) running at 1667MHz.

4.1 Overhead of MrsP Primitives

A first important metric is the cost of the primitives involved in the use of MrsP resources. Such overhead gives a threshold under which it is not convenient to use the protocol: if the use of the resource is less than the overhead of the protocol, a simple non-preemptive section is a better approach. The overhead we report is by no means an absolute value (since it depends on both the OS and the hardware), but it can give an idea of the general cost of using MrsP resources. Results are summarized in Table 1. All the experiments were performed in the scenario where resources are not nested.

Table 1. Overhead of conceptually similar primitives in RTEMS and LITMUS^{RT}

RTEMS		LITMUS^{RT}	
`obtain`	$5,376$ *ns*	$8,800$ *ns*	`lock`
`release`	$5,514$ *ns*	$8,500$ *ns*	`unlock`
`ask for help`	$1,827$ *ns*	$35,000$ *ns*	`finish switch`

Results on RTEMS. In RTEMS there are three main procedures in which MrsP performs its work: *obtain resource*, *release resource* and *ask for help*. The experiments are performed incrementing up to 23 the number of tasks that act as rivals on a specific resource (one task per partition). The thread dispatch was intentionally disabled in order to evaluate the cost of the procedure while avoiding the cost of preemptions and migrations.
Obtain resource updates the necessary data structures (e.g., raise the priority of the task to the ceiling) and, if necessary, initializes the MCS lock. The maximum observed value is $5,376$ *ns*.
Release resource: it restores the state of the used data structures (e.g., restore the priority to the task) and, if necessary, updates the resource tree to point to the next resource holder. The maximum observed value is $5,514$ *ns*.
Ask for help looks inside the resource tree (since there is no nesting, the tree equals a list) and finds a possible spinning task that is available to help. The worst case is met when all 23 threads are queued but no one of them is actually spinning, and its maximum observed value is $1,827$ *ns*.

The release resource and help procedures are subjected to variability. In fact, both these procedures need to inspect the resource tree: the bigger it is, the costly the primitive. A valid upper bound for these procedures depends on the available number of processors, that is, the maximum degree of parallelism available.
Results on LITMUS^{RT}. MrsP operates using three LITMUS^{RT} primitives: *resource lock*, *resource unlock* and *finish switch*.
Resource lock manages the ceiling and the FIFO queue in 800 *ns*, when it is necessary to yield the processor to the resource holder it updates the remote partition and perform the migration in $8,000$ *ns*, and each spinning cycle costs 500 *ns* (maximum delay to stop spinning).
Resource unlock restores priorities and releases the resource in 500 *ns*; the operations on remote processors, to migrate back or to migrate the new resource holder in another partition (when necessary) cost $8,000$ *ns*.
Finish switch operates under different scenarios to enforce the MrsP invariant. If the lock holder is preempted and there is at least one running task waiting for

the resource, the task migrates to that partition. This requires to search for a spinning task and to perform a migration, for a cost of $35,000$ ns. If the resource owner migrated to a remote processor but it is not running any more, then it needs to migrate back. The notify mechanism involved requires $6,000$ ns.

Part of the observed overheads are intrinsic to the implementation of MrsP and they reflect the need to share the data structures of the protocol. The high cost of migrations (an average of $6,000$ ns per migration) is to be attributed to different sources (e.g., the operations that Linux uses to enforce consistency before and after a migration, the internal state of the partitions). Our experiment shows anyhow that in absence of migrations the protocol adds a small overhead: in the average case where no migration is needed to obtain or release a resource, the protocol overhead is less than 1 μs.

4.2 Intrusiveness of MrsP

This second experiment evaluates the net cost of the MrsP framework when no MrsP resources are actually used. The experiment highlights how much the implementation of MrsP must interact with normal scheduling operations.
Results on RTEMS. Figure 3 shows the maximum observed time to perform three of the main procedures used by the FP scheduler in RTEMS while the MrsP protocol is or is not implemented. The simple fact of having available MrsP inside RTEMS even while not using it, causes an overhead (approximately 100 instructions) to the main scheduling procedures. This stems from the need to modify the internals of the scheduler to correctly manage the migration of tasks (in an environment that normally does not permit it) as well as to perform part of the checks required to enforce the MrsP invariant.
Results on LITMUS$^{\mathbf{RT}}$. Figure 4 shows the maximum observed time to perform the main scheduling primitives in the partitioned fixed-priority scheduler (PFP): schedule, job release and finish switch. The experiment reveals that the integration of MrsP has a low impact when not in use, despite the support required by the helping mechanism: the complexity of the Linux kernel far outweighs the operations needed to enforce the MrsP invariant.

4.3 Cost of Supporting Nested Resources

As highlighted in Section 3.1, the implementation of MrsP in RTEMS uses a resource tree. Since the tree grows dynamically and its size is proportional to the number of tasks partaking in the protocol, and since it is scanned in order to find tasks available to help, we performed and experiment to understand how much overhead the use of the tree induces. Each sampled value in Figure 5 represents the maximum time that it takes to look inside a full tree of height x, with no task available to help. Each level of the tree contains 23 tasks.

It comes without surprise that the overhead induced by the inspection of the resource tree is linear to its size. This shows an implementation flaw in the MrsP implementation of RTEMS. Unrelated high priority tasks can suffer from excessive resource nesting of unrelated lower priority tasks. A possible

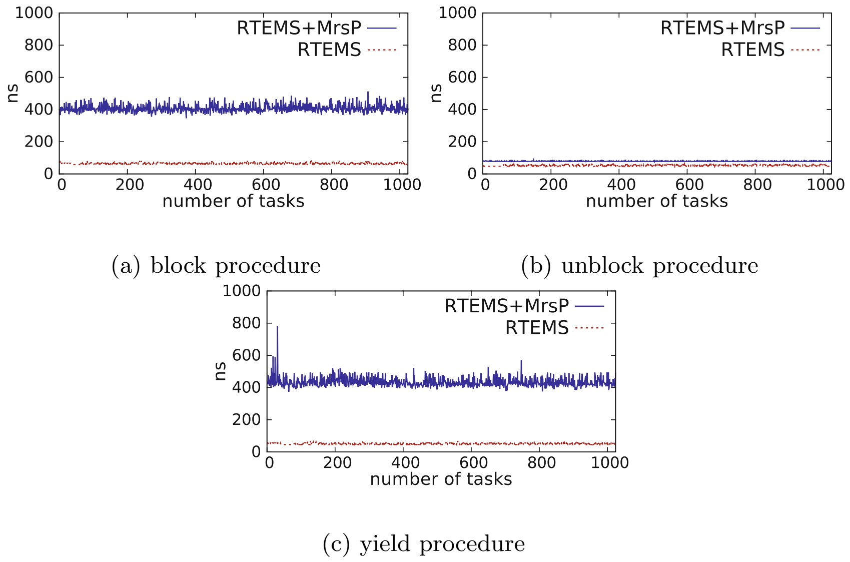
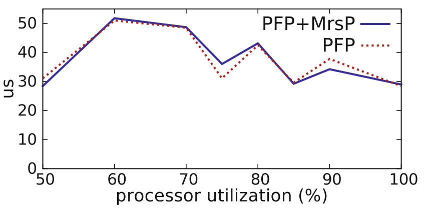


Fig. 3. Max observed behavior of relevant primitives on RTEMS

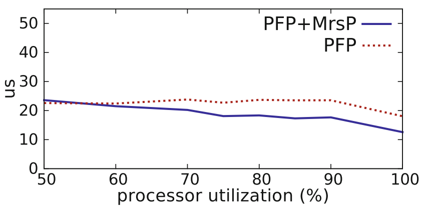
improvement would be to pre-compute the highest priority tasks available for help in each partition. This would limit the cost of the search by the number of partition in the system. However, the obtain and release operations would be more expensive since such information must be kept updated.

5 Related Work

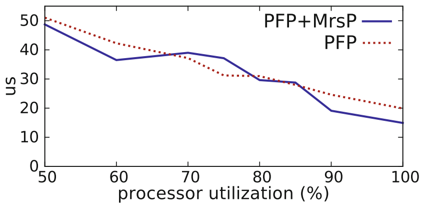
A large majority of state-of-the-art multiprocessor resource sharing protocols were originally conceived as an extension of well-known uniprocessor techniques to a new scenario where resource accesses can happen in parallel and task priorities (used by ceiling-based protocols) are not comparable when tasks belong to different partitions. Simpler multiprocessor protocols [22], [19] use some kind of *priority boosting* mechanism to speed up the use (and release) of global resources and to reduce the time spent by tasks waiting for remotely locked global resources. Concepts that are very similar to priority boosting are also exploited by more advanced approaches, e.g.,[17], [9], [5]. The main drawback of priority boosting mechanism and its derivatives, however, is that they indiscriminately interfere with all local tasks. A possible solution to this problem has been identified in the use of a *helping mechanism*: a mechanism that lets tasks waiting on a global resource "help" (take care of the execution of) a remote preempted resource holder. A helping mechanism is used, for example, in the Multiprocessor Bandwidth Inheritance Protocol (M-BWI) [15], in the $O(m)$ Independence-preserving Protocol (OMIP) [7] and in the Server Based Locking Protocol (SBLP) [6]. All these approaches, however, do not lend themselves to an easy integration in the classic RTA framework, which instead MrsP does.



(a) schedule procedure



(b) job release procedure



(c) finish switch procedure

Fig. 4. Max observed behavior of relevant primitives on LITMUS^{RT}

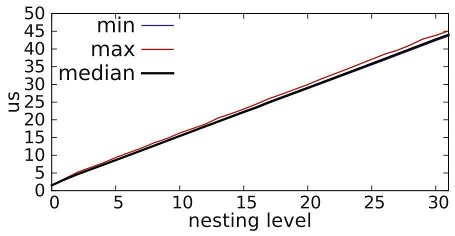


Fig. 5. Overhead induced by traversing the whole resource tree. Each level of the tree has 23 elements: the size of the tree is $(x + 1) * 23$.

To the best of our knowledge, only few works in literature address the problem of implementing and evaluating a multiprocessor resource sharing protocol with a strong emphasis on low-level design and implementation issues. The work in [8] offers two main contributions: an improved analysis to accurately account for the blocking contributions of several multiprocessor resource sharing protocols (MPCP[19], DPCP[22] and FMLP+[5], among others) and an extensive empirical evaluation of such protocols, spanning from their algorithmic comparison to the evaluation of their implementations in LITMUS^{RT}. The evaluation is however limited to the *lock* and *unlock* procedures, since the studied protocols make no use of a helping mechanism. Consequently, the induced overhead on other scheduling procedures is almost null. The same work also discusses the algorithmic principles for multiprocessor resource sharing protocols, giving some coarse grained guidelines about queueing resource requests (either FIFO or priority ordered) and the opportunity of executing critical sections through remote agents or locally.

The same protocols have been also evaluated in a previous work [10], whose main goal was to describe their implementation on top of LITMUS$^{\text{RT}}$. A first contribution in [10] consists in the exploration of a number of design issues, arising from practically implementing non-trivial resource sharing protocols. In contrast with our work, such issues are largely related to the provision of an efficient and robust support for resource sharing in LITMUS$^{\text{RT}}$ (e.g, where to store the priority for inheritance/ceiling protocols, how to generalize the Linux wait queues to allow ordering elements). The implemented protocols were also compared based on their runtime performances. An important conclusion drawn from the authors is that the algorithmic performance of a multiprocessor resource sharing protocol dominates its runtime performance: the general overhead induced from the *lock* and *unlock* procedures is small compared to the advantages that stem from the determinism that the use of such protocols give.

The work in [16] deals with the construction, implementation and evaluation of M-BWI protocol. Similar to the work in [10], the discussion on the implementation of the M-BWI protocol focuses on explaining the specific solution used by the authors to adapt the protocol inside LITMUS$^{\text{RT}}$ without exploring other possible designs, but still exposing (hidden) corner cases that must be addressed to soundly implement the protocol. The evaluation of M-BWI that the authors propose in their work, share similar traits with our work. Similar to our evaluation, they compared the cost of the three main primitives involved in the use of global resources (i.e., *schedule*, *lock* and *unlock*) when (i) the system does not support the protocol, (ii) the system support the protocol but no global resource is used, and (iii) global resources are used. Their results depict the same trend of our implementation of MrsP in LITMUS$^{\text{RT}}$: the overhead induced by the protocol in the scheduling decision, besides the *lock* and *unlock* primitives, is negligible as compared to the cost induced by the Linux kernel primitives.

6 Conclusions

With this work we provide evidence that MrsP can effectively be implemented on standard RTOS. In our implementation effort we identified and addressed some design issues, spanning from the data structures to be used, to the management of particularly subtle corner cases in the scheduling operations. Arguably, these implementation issues were not explicitly in the original formulation of MrsP [11] because they are strictly coupled with the kernel support and structures provided by the specific RTOS.

We also performed some experiments to evaluate the runtime behavior of the protocol. The overheads incurred by our implementation of MrsP are generally acceptable, assuming critical sections of non-negligible length. A limited increase in the execution time of the kernel primitives is compensated by the improvement in the response time of tasks sharing global resources, with no interference on independent tasks [11]. In future work, we are interested in further analyzing the various contributions (in terms of kernel overhead) that MrsP induces at runtime. We are particularly interested in evaluating the costs of job migration. Finally, we aim at taking all the overheads into account within the RTA framework.

References

1. Abella, J., Cazorla, F., Quinones, E., Grasset, A., Yehia, S., Bonnot, P., Gizopoulos, D., Mariani, R., Bernat, G.: Towards improved survivability in safety-critical systems. In: 17th IEEE International On-Line Testing Symposium (IOLTS) (2011)
2. Anderson, T.: The performance of spin lock alternatives for shared-money multiprocessors. IEEE Transactions on Parallel and Distributed Systems (1990)
3. Andersson, B., Jonsson, J.: The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In: 15th Euromicro Conference on Real-Time Systems (ECRTS) (2003)
4. Baker, T.: A stack-based resource allocation policy for realtime processes. In: 11th IEEE Real-Time Systems Symposium (RTSS) (1990)
5. Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: A flexible real-time locking protocol for multiprocessors. In: 13th IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA) (2007)
6. Bonato, L., Mezzetti, E., Vardanega, T.: Supporting global resource sharing in RUN-scheduled multiprocessor systems. In: 22nd International Conference on Real-Time Networks and Systems (RTNS) (2014)
7. Brandenburg, B.: A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In: 25th Euromicro Conference on Real-Time Systems (ECRTS) (2013)
8. Brandenburg, B.: Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In: 19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, pp. 141–152 (2013)
9. Brandenburg, B., Anderson, J.: Optimality results for multiprocessor real-time locking. In: 31st IEEE Real-Time Systems Symposium (RTSS) (2010)
10. Brandenburg, B.B., Anderson, J.H.: An lmplementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In: 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, pp. 185–194 (2008)
11. Burns, A., Wellings, A.: A schedulability compatible multiprocessor resource sharing protocol - MrsP. In: 25th Euromicro Conference on Real-Time Systems (ECRTS) (2013)
12. Calandrino, J., Leontyev, H., Block, A., Devi, U., Anderson, J.: LITMUS^{RT}: a testbed for empirically comparing real-time multiprocessor schedulers. In: 27th IEEE Real-Time Systems Symposium (RTSS) (2006)
13. Davis, R., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4) (2011)
14. Edelin, G.: Embedded systems at THALES: the artemis challenges for an industrial group. In: Invited Talk at the ARTIST Summer School in Europe (2009)
15. Faggioli, D., Lipari, G., Cucinotta, T.: The multiprocessor bandwidth inheritance protocol. In: 22nd Euromicro Conference on Real-Time Systems (ECRTS) (2010)
16. Faggioli, D., Lipari, G., Cucinotta, T.: Analysis and implementation of the multiprocessor bandwidth inheritance protocol. Real-Time Systems **48**(6), 789–825 (2012)
17. Gai, P., Natale, M.D., Lipari, G., Ferrari, A., Gabellini, C., Marceca, P.: A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In: 9th IEEE Real-Time and Embedded Technology and Applications Symposium (2003)

18. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. (1979)
19. Lakshmanan, K., De Niz, D., Rajkumar, R.: Coordinated task scheduling, allocation and synchronization on multiprocessors. In: 30th IEEE Real-Time Systems Symposium (RTSS) (2009)
20. LITMUS^{RT}. http://www.litmus-rt.org/
21. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (1991)
22. Rajkumar, R.: Synchronization in Real-Time Systems: A Priority Inheritance Approach. Kluwer Academic Publishers (1991)
23. RTEMS. http://www.rtems.org/