Supporting Nested Resources in MrsP

Jorge Garrido[1], Shuai Zhao[2], Alan Burns[2], and Andy Wellings[2]

[1] Sistemas de Tiempo Real e Ingeniería de Servicios Telemáticos (STRAST)
Universidad Politécnica de Madrid (UPM)
[2] Department of Computer Science, University of York

Abstract. The original MrsP proposal presented a new multiprocessor resource sharing protocol based on the properties and behaviour of the Priority Ceiling Protocol, supported by a novel helping mechanism. While this approach proved to be as simple and elegant as the single processor protocol, the implications with regard to nested resources was identified as requiring further clarification. In this work we present a complete approach to nested resources behaviour and analysis for the MrsP protocol.[3]

1 Introduction

Both the increasing requirements in terms of computation power and the decreasing availability of single processor platforms have given rise to the need for safe, analysable real-time multiprocessor systems. While providing more execution units increases the overall computation power, it also increases the complexity of the required scheduling protocols with regard to shared resources and task communication management.

Single core approaches benefited from the inherent serialization on access requests imposed by the existence of only one processor. Multicore approaches to shared resources have explored different ways of providing a bound on the time it takes to gain access to such resources. One of the main approaches is to use spin-locks. Following this approach, a task requesting access to a resource places the request on a queue and spin-waits at a certain priority until the access request is satisfied. If the synchronisation protocol does not allow higher priority tasks to preempt tasks accessing shared resources then higher priority tasks may suffer unnecessary blockings. Alternatively, if access requests can be preempted, then a mechanism has to be defined to ensure progress on the locked resource if other tasks are blocked by a resource held by a locally preempted task. This last approach is the one followed by the Multiprocessor Resource Sharing Protocol (MrsP) [4]. In this protocol a helping mechanism is defined, by which locally preempted tasks can migrate to other processors to make progress provided that a task is actively waiting on that processor to access the locked resource.

In this paper we analyse the life cycle of a task with regard to the MrsP shared resource protocol and define a set of rules supporting a fine grained analysis for preemptive, FIFO spin-lock controlled, nested resources.

[3] This work has been partially funded by the Spanish National R&D&I plan (project M2C2, TIN2014-56158-C4-3-P).

2 Related work

Despite the academic interest in multiprocessor real-time systems, many proposals are oblivious to, or explicitly ban, task communication and synchronization. Among the work on shared-memory synchronization protocols for multiprocessors real-time systems published up to date, few publications address the analysis of nested resources as required by the complex paradigms of synchronization required by modern real-time systems.

A common approach to supporting nested resources has been to group resources together. In this approach, of which FMLP [2] is a notable example, nested resources are locked and released as a whole. This unfortunately seriously undermining the concurrency of the system, thereby reducing schedulability.

The first proposal for fine-grained analysis was proposed in [10]; forcing a strict order on locks and releases (locking operations are not allowed after a release has been performed on the nesting). An extension of this work is the Real-time Nested Locking Protocol (RNLP) [12,13] which limits the concurrency on nested resource accesses by means of a token mechanism and provides a set of request satisfaction mechanisms aiming for optimality under different system configurations.

Recent work has provided a fully fine-grained blocking bound for nested non-preemptive FIFO spin locks under partitioned fixed-priority scheduling [1]. This is achieved using a novel graph abstraction of the blocking interaction among tasks and resources for which, given a set of invariants stating graph properties, an Integer Linear Programming (ILP) approach is used to find a subgraph yielding a safe worst-case blocking value.

It is also worth to mention SPEPP [11] as a relevant protocol introducing the notion of a helping mechanism, fundamental to MrsP formulation. This helping mechanism was also used in M-BWI [7,8] to deal with the issue of tasks running out of budget while holding a resource in systems ruled by execution-time servers.

Despite the fact that MrsP was recently proposed, it has been effectively implemented [6] in Litmus [5] and RTEMS [9]. Its implementability in Ada is discussed in [3], where a prototype outside-kernel implementation is presented.

3 System and task model

The baseline of the current work is the MrsP proposal [4]. In this work a protocol to provide a safe upper bound to resources shared among tasks potentially executing on different processors is presented. In this work, the general *sporadic task model* is considered, under fully partitioned systems. Deadlines are unconstrained, but there can not be more than one active job of a task at a time. As such, the terms task and job are used interchangeably in this paper. Resources are required to be accessed under mutual exclusion. Preemptive fixed-priority scheduling is assumed.

Tasks are related to resources by means of different functions: $G(r^j)$ is the set of tasks that access directly a resource r^j, and $F(\tau_i)$ returns the set of resources

used by task τ_i; function *map* returns the set of processors where the argument entities execute, and || returns the size of a set. If all tasks that access a resource execute on the same processor then the resource is deemed to be *locally accessed*, otherwise it is contended for *globally*.

MrsP, in general, follows the rules of the Priority Ceiling Protocol (PCP): resources are given a Local Ceiling Priority on each processor which is equal to the highest priority of any local task that accessing the resource. Tasks, when attempting to access a resource, rise their active priority to that Ceiling Priority. Authors in [4] claim to inherit four fundamental properties from PCP:

- A job is blocked at most once during its execution.
- This blocking takes place prior to the job actually executing.
- Once a job starts executing, all the resources it needs are (locally) available.
- Deadlocks are prevented.

The scheduling analysis for MrsP keeps the form of Response-Time Analysis (RTA) as in the PCP case, defined in the following equation:

$$R_i = C_i + max\{\hat{e}, \hat{b}\} + \sum_{\tau_j \in \mathbf{hpl}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1}$$

where R_i is the worst-case response-time of task τ_i, $\hat{e}$ is the maximum arrival blocking due to local lower priority tasks accessing shared resources, and $\hat{b}$ is the maximum non-preemptive execution time caused by the underlying OS/kernel. C_i is decomposed into the Worst Case Execution Time (WCET) of the task outside its use of shared resources plus the cost of accessing (e) each shared resource r up to n times during each activation:

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

Finally, e is calculated as the cost of each individual access, c^j, multiplied by the number of processors from where the resource can be accessed (this is the maximum length of the FIFO queue):

$$e^j = |map(G(r^j))| c^j \tag{3}$$

This safe upper bound to the access cost is based on two properties of MrsP:

- Only one task per processor can be accessing a resource at any given time. This is directly inherited from PCP.
- A helping mechanism, proposed in [4], by which tasks spin-waiting to access a resource can take over the execution of tasks locally preempted while holding the required resource.

Since the helping mechanism is the most relevant and novel feature in [4], and highly influences the behaviour of the system, it will be further explained in the rest of this section.

Figure 1 represents the different logical states in which a task can be with regard to MrsP controlled resources:

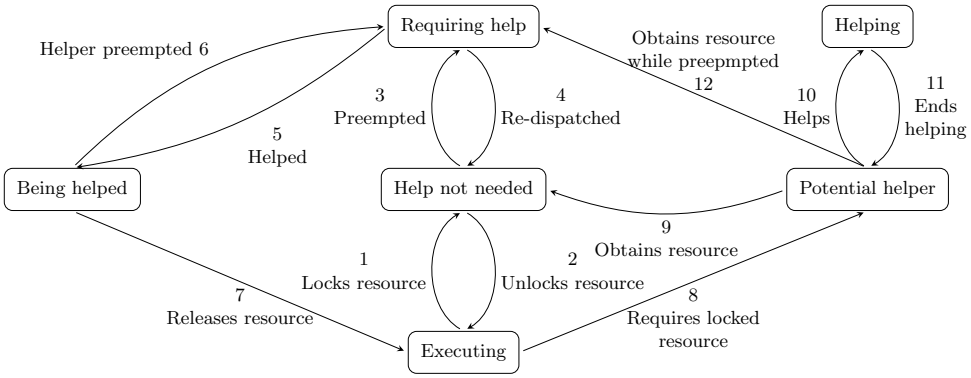

Fig. 1. Task state diagram of helping mechanism without nested resources.

- *Executing*: A task that does not require any resource to make progress.
- *Help not needed*: A task is making progress with a locked resource while being dispatched on its host processor by means of its active priority.
- *Requiring help*: A task holding a global resource that is unable to make progress (as it has been locally preempted) from its host processor.
- *Being helped*: A task that holds a global resource and has migrated to another processor in order to make progress.
- *Potential helper*: A task that requests an already allocated resource, and is spin-waiting for it.
- *Helping*: A task that was spin-waiting and pulled a requiring-help task to make progress on its processor in order to help it to release the requested resource.

Every task initially holds no resource, so its in the *executing* state. At a certain point, a task can request access to a global shared resource. As part of the process of this request, it increases its active priority to the Local Ceiling Priority of the resource. If the resource is free, it will lock the resource (transition, or tran, 1). Otherwise it will be spin-waiting blocked by this resource until access is granted to the resource (tran 8).

Transition 1, locking the resource, moves the task to the *help not needed* state. While in this state, the task can: finish the access to the resource and release its associated lock (tran 2), or be locally preempted while accessing the resource (tran 3).

If a task is locally preempted while holding a lock, it is considered to *requiring help* to make progress on the resource. While it remains in the *requiring help* state, no progress is possible. If no other task requires the locked resource, while being preempted, then this preemption time is just local interference, and the *requiring help* task will, at some point (when the preempting job terminates), be re-dispatched at its host processor due to its active priority (tran 4).

However, if at some point while being preempted, another task requests access (or was already spin-waiting) to the resource, this task will help the preempted

one (tran 5 for the preempted task). This transition, in practice, implies a migration to the helper host processor, with the active priority updated to the Local Ceiling Priority of the held resource on that processor. Then, the task will make progress (*being helped*) until it releases the resource, migrating back to its host processor with its base priority (tran 7), or until it is preempted again on the helping processor, *requiring help* (tran 6) again until it is re-dispatched on its own processor or is helped again.

Tasks blocked by a locked resource are *potential helpers*. Their request is added to a FIFO queue and will be served when all requests in front have been satisfied. This can happen when the task is actually spin-waiting for the resource (tran 9), immediately making progress on the resource, or when the task is locally preempted. As it would hold a resource without making progress due to being locally preempted on its host processor, it would be considered to be *requiring help* (tran 12).

If, while being a *potential helper* due to being blocked by a locked resource, the holder of that resource is locally preempted and thus *requires help*, the helping mechanism is fired. This, in practice means that the *potential helper* task pulls the *requiring help* task to its host processor and lends it its active priority, to execute on its behalf (tran 10). The *helping* procedure ends when the helped task releases the held resource or it is preempted on the helping processor (tran 11).

Thus, for a task to be helped, there should be both a task *requiring help* and a *potential helper* for the same resource. The helping mechanism begins with transition 5 for the *requiring help* and transition 10 for the *potential helper*. Equivalently, the helping mechanism ends with a helped task transitioning by 6 or 7, and a helper doing transition 11.

While these behaviours deal adequately with non-nested resources requests, systems including nested resources require a more specific approach. The full description and definition of such an approach is the main contribution of this paper.

4 Nested resources

The system model and analysis presented in [4] and briefly summarized in section 3 can not, by themselves, be transferred to a system with nested resources. Equation 3 only reflects direct accesses from tasks to resources. In [4], a new term, $V(r^j)$ was proposed as a function returning the set of resources accessing the resource r^j. Based on that definition, the following equation for calculating the cost of accessing a nested resource was proposed:

$$e^j = (|V(r^j)| + |map(G(r^j))|)c^j \tag{4}$$

Equation 4 now defines the maximum queue length for accessing the resource as the number of processors from where the resource can be directly accessed plus the number of outer resources from where the resource can be accessed. While this interpretation of the queue length is correct, the value of e^j does not

necessarily represent a safe upper bound for a resource that requires accessing inner resources to complete its execution. The reason for this is that the analysis fails to account for the possible transitive blocking while accessing that inner resources (r^k). That is, a task (or outer resource) attempting to access resource r^k may find it already locked, and be unable to make progress. We shall produce a correct version of this equation in section 4.2.

Another issue raised when considering nested resources in MrsP is local blocking. For non-nested resources, it is proven that, following PCP behaviour, a task can only be blocked once, and only before it actually gets to execute. This property is necessary to maintain the $max\{\hat{e}, \hat{b}\}$ factor for the local blocking in equation 1. However, the helping mechanism proposed in [4], together with nested resources, could lead, if no measures are taken, to situations in which higher priority tasks can be blocked more than once after beginning their execution. Due to the helping mechanism, a task holding a resource and locally preempted can be migrated to another processor, in order to make progress, While migrated, it might lock an inner local resource with a higher priority. If the active priority of the task is raised then, on return to the host processor it would preempt a higher base priority task thus causing further delayed local blocking. For this reason the active priority of a migrated task is not raised in this situation.

4.1 Desired nested resource behaviour

In this work we propose a complete approach to global nested shared resources, providing a safe upper bound access cost for nested resources as well as a dynamic priority assignment scheme preserving PCP properties. Figure 2 depicts

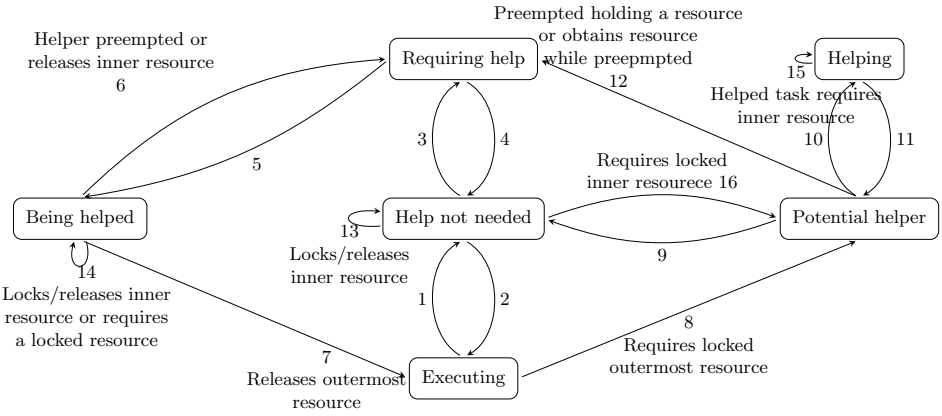

Fig. 2. Task state diagram of helping mechanism with nested resources.

the different logic states of tasks under MrsP, when considering nested resources.

While the states remain the same, new transitions arise and some existing ones are now triggered by new events.

Tasks still begin *executing* without any shared resource, and transitions 1 and 8 are triggered when the task requires the outermost resource of a nested call, raising the active priority to the ceiling of that outermost resource. If the access request is satisfied immediately, the task executes without requiring any help. While executing in the *help not needed* state all locks and releases update the active priority of the task (tran 13) as in PCP. If a lock request finds a resource already locked, the task updates its priority to the local Ceiling Priority of the resource and becomes a *potential helper* for that resource (tran 16).

As with the non nested case, the task can, while executing not being helped, be locally preempted and thus *require help* to make progress (tran 3). If at some point while *requiring help*, another task is spin-waiting for one of the resources locked by this preempted task, it will be helped by the spinning task. However, in the nested case, the helper may be helping not due to requesting the innermost locked resource, but due to requesting any of the resources held by the preempted task.

A task, when migrated to be helped (tran 5), is granted the priority of the helper task. While *being helped*, a task is allowed to lock and release further resources (tran 14), but these actions *do not change the priority of the helper*, and thus the priority at which the helped task is executing while *being helped*.

As with the non nested case, a task can, while *being helped*, release its outermost locked shared resource and migrate back to its host processor with its base priority (tran 7). Similarly, a task can leave the *being helped* state to *requiring help* (tran 6). In the nested case, this transition can be triggered by both the task being preempted on the helping processor, and by releasing of the required nested resource by the helping task. In this latter case, the task *being helped* still holds other resources, and still *requires help* to make progress.

Any task finding a required resource already remotely locked while *executing* or in *help not needed* state becomes a *potential helper* for that resource (trans 8 and 16). While being a *potential helper* a task can be preempted. In this case, if the task holds a resource, it *requires help* to make progress on that resource (tran 12).

Potential helpers are ready to help tasks *requiring help*, holding their required resource (tran 10). A task while *being helped* may require a locked inner resource. In this situation, the task is still considered to be helped (tran 14) and spin-waits for locked resource. If the third task holding that inner resource is also *requiring help*, the helping task is ultimately blocked by this third task not making progress. As such, the helper task will also help the third task migrating it to its host processor, and giving it its active priority, executing instead of the task that was *being helped* before (tran 15). This transitive help is maintained until the third task releases the inner resource required by the original helped task.

The helping mechanism can end (tran 11) due to the same two reasons as in the non nested case: the resource required being released or the helper task being locally preempted, with the same implications as in the non nested case.

4.2 Updated analysis and properties

In this subsection we propose an analysis in which a safe upper bound can be obtained for the access cost to a resource including any of the inner resources required by this resource. To provide such analysis, we require a strict irreflexive partial order on the resource nesting. This not only prevents deadlocks, but also provides an end to the recursion in the analysis, as at least there has to be one resource in the system not requiring any other resource to complete its execution. Given this, the access cost for a nested resource is now defined as follows:

$$e^j = (|V(r^j)| + |map(G(r^j))|) * (c^j + \sum_{r^k \in \mathbf{U}(r^j)} n_j^k e^k) \tag{5}$$

where $\mathbf{U}(r^j)$ is the set of inner resources directly accessed by r^j and n_j^k is the number of times an inner resource r^k is accessed on each access to r^j.

In equation 5, the length of the queue is as in [4], where PCP limits the number of concurrent access attempts to a resource to one at a time per processor ($|map(G(r^j))|$) and the mutual exclusion nature of shared resources under MrsP ensures that only one access attempt can be performed at a time from any outer resource, giving the total number $|V(r^j)|$. Note this queue length may be pessimistic, but our objective here is to provide sufficient analysis.

For the cost of the access itself, now we do not only consider the cost of the accessed resource itself but the cost of accessing all the nested resources. So e^j now represents the full cost for a task accessing nested resources via r^j as an outermost resource, or the cost for outer resources accessing e^j and all its inner resources.

This way of calculating the e value for nested resources now includes the possible transitive blocking on each access. As each access is not considered isolated, but includes the cost of inner resources queues (which are the source of transitive blocking), now equation 5 provides a safe upper bound.

Considering the extra blocking a task may suffer due to the helping mechanism, in this proposal we require that a task does not update (increase or decrease) its host active priority while being helped. This way, lower priority tasks can not benefit from the helping mechanism to increase their priority while migrated, with the undesired side effect of causing extra blocking to local higher priority tasks. In turn, tasks are dispatched on their host processor with the priority they had when they were locally preempted. We will refer to this priority as the *Leaving Priority* for the rest of the paper. Migrated tasks *do* update their active priorities when they are re-dispatched on their host processor.

Example. To illustrate the approach, the example for nested resources analysis presented in [4] is now revisited. Consider a system with four tasks, τ_1, ... , τ_4,

executing on four different processors $p_1, ..., p_4$, and two resources, r^1 and r^2, with execution times c^1 and c^2 respectively. Tasks τ_1 and τ_2 access r^1 directly, and τ_3 and τ_4 access r^2 directly. In addition r^1 accesses r^2, so, for example, when τ_1 accesses r^1 it will, while holding r^1 also access r^2.

Table 1. Task allocation and resource usage.

Task	Processor	$F(\tau_i)$
τ_1	p_1	r^1, r^2
τ_2	p_2	r^1, r^2
τ_3	p_3	r^2
τ_4	p_4	r^2

Resource	$G(r^i)$	$V(r^i)$	$map(G(r^i))$
r^1	τ_1, τ_2	$\emptyset$	p_1, p_2
r^2	τ_3, τ_4	r^1	p_3, p_4

As presented in Section 4.2, the nested resource analysis proposed is solved by iteration from inner to outer resources. In this example, we have one inner resource, r^2, and one outer resource, r^1. The accessing cost of the inner resource is (following equation 5):

$$e^2 = (1 + 2) * (c^2) = 3c^2$$

Then we can calculate the cost of accessing the nesting of resources via r^1, as we know the cost of accessing all its inner resources (r^2):

$$e^1 = (0 + 2) * (c^1 + e^2) = 2(c^1 + e^2) = 2(c^1 + 3c^2)$$

Now e^1 is a safe upper bound, including transitive blocking, for the access to r^1 and all its required inner resources. We note an incorrect answer is given for this example in [4].

4.3 Improved nested helping analysis

With the current definition of local and global resources ceiling priorities , there are situations in which the analysis can benefit from other priority assignments. Specifically, resources accessed only by tasks allocated to the same processor via outer global resources receive a pessimistic analysis. This pessimism can be reduced and in some cases eliminated by a combination of a particular priority assignment (giving global resources encapsulating a call to an inner local resource the ceiling priority of this inner local resource) and the definition of an equivalent task set reflecting the behaviour of the system with that particular assignment of priorities.

Consider a system comprising a specific processor P_1 with a task set including, among others (irrelevant for the example) the following tasks: tasks τ_1, τ_2, τ_3 with lowest priorities on P_1, and τ_{10} with the highest priority on P_1. On this processor, there is a set of local resources r_l^1, r_l^2, r_l^3, which are only accessed by tasks τ_1, τ_2, τ_3 and τ_{10}. Task τ_{10} accesses the local resources directly, while

τ_1, τ_2, and τ_3 do so via a global resource, different for each of them. These resources are accessed only from tasks from P_1 and another processor, but accesses from the other processor do not generate accesses to r_l^1, r_l^2 and r_l^3. The relevant information for the example is summarized in table 2.

Table 2. Task allocation and resource usage without improvement.

Task	Processor	$F(\tau_i)$
τ_{10}	P_1	r_l^1, r_l^2, r_l^3
τ_3	P_1	$r^3 \rightarrow r_l^3$
τ_2	P_1	$r^2 \rightarrow r_l^2$
τ_1	P_1	$r^1 \rightarrow r_l^1$

Resource	$G(r^i)$	$V(r^i)$	$map(G(r^i))$
r^1	τ_1, τ_1'	$\emptyset$	P_1, P_2
r^2	τ_2, τ_2'	$\emptyset$	P_1, P_3
r^3	τ_3, τ_3'	$\emptyset$	P_1, P_4
r_l^1	τ_{10}	r^1	P_1
r_l^2	τ_{10}	r^2	P_1
r_l^3	τ_{10}	r^3	P_1

Given the analysis presented in table 2, the access cost for the highest priority task τ_{10} of each local resource would be (considering execution times of global resources c_g and local resources c_l): $r_l = 2c_l$, being the total access cost for the three resources $r_l^{1,2,3} = 3 \cdot 2c_l$. This analysis assumes that the higher priority task may have to wait for the lower priority tasks on each access to the local resources. This is due to the access of the lower priority tasks via a global resource. If this was not the case, r_l^1, r_l^2 and r_l^3 would be pure local resources and be completely ruled by PCP. As the lower priority tasks can be preempted while holding the global resources, and each of them can migrate to a different processor to make progress, the three of them can access their respective local resource concurrent with τ_{10} while being helped remotely. In this case, the helping mechanism produces a high blocking time for a high priority task accessing directly to local shared resources. This clearly contradicts the aim and intuition behind PCP and MrsP.

This problem can be addressed by reducing the concurrency of the lower priority tasks. If r_l^1, r_l^2 and r_l^3 are given the same local Ceiling Priority then only one of the three tasks τ_1, τ_2, or τ_3 can gain access to their outer resource. As a result only one can be helped, and only one can gain access to the inner resource while migrated. The impact on τ_{10} is reduced to a single block.

5 Definitions

The detailed approach for MrsP systems supporting nested resources is now presented as a set of rules, lemmas, properties and theorems. Those from PCP and non nested MrsP are assumed and hold unless overridden by those presented here.

Rule 1. Resources under MrsP nest following a strict irreflexive partial order.

Rule 2. A task being helped executes on the helper processor with the helper active priority.

Rule 3. The helping mechanism can be initiated due to the helper task requesting access to any of the resources held by the helped task.

Rule 4. The helping mechanism is transitive, *i.e.* a helper task shall help the locally preempted task ultimately preventing it from making progress.

Property 1. A task holding one or more resources, that is not being blocked accessing another resource, will make progress if there is a task spin-waiting due to being blocked by any of the resources held.

This is the fundamental novel property from MrsP that we wanted to move to nested resources, as it provides the safe upper bound expressed by equation 5.

Rule 5. Task only modify their active priority when they are dispatched in their host processor, not being help.

Tasks can lock and release resources whenever they are executing. If they do so while not being helped, the active priority of the task is modified according to PCP rules. If they do so while being helped, there is no modification of active priorities of any the helping or the helped task. The helped task will update its active priority according to the resources held when it is dispatched again on its host processor when its leaving priority is the highest among the tasks eligible to execute.

Rule 6. The helping mechanism shall also be conducted between tasks allocated to the same host processor.

Rule 7. Tasks remain notionally eligible to be dispatched (at their leaving priority) on their host processor while being helped.

By considering tasks being helped and executing on another processor as eligible for dispatching on their host processor, lower priority tasks are prevented from executing when a higher priority task would be executing instead.

Lemma 1. A task is only allowed to begin its execution if all higher base priority tasks allocated on that processor are completed.

Proof. If no task is migrated, then all uncompleted tasks are ready to execute on that processor. Following PCP rules, the task dispatched is the one with higher active priority. For a task that has not locked any resource, all higher base priority task have higher active priorities. As a task can not have locked any resource before actually executing, it is proven. If a higher base priority has migrated, its leaving priority is at least equal to its base priority. Then, following Rule 7 the higher priority task would be eligible to execute against any lower base priority task not holding any resource. □.

Lemma 2. A task can only make progress by being helped if there is a pending task on the same host processor with higher base priority than its active priority.

Proof. A task having a lower priority than the base priority of another task, will keep having a lower priority unless it locks a resource. Given PCP rules and Rule 7, the lower priority task can not be dispatched on its host processor with that lower priority until the higher base priority task is completed. Until then, it can only lock another resource while making progress because of helping. Due to Rule 5, this will not increase its active priority, this will keep it below the

higher base priority of the pending task. As a result, a task can not be executed if not being helped while there are higher base priorities pending tasks. □.

Corollary. Tasks with lower active priorities than pending tasks with higher base priorities can not increase their active priority.

Proof. Proven during proof of Lemma 2. □.

Lemma 3. Each task can suffer at most a single local block per activation, and this blocking occurs before the task actually executes.

Proof. Lemma 3 in [4] proves this property for MrsP without nested resources, based on the properties of PCP. For nested resources, as tasks are not allowed to increase their priority while migrated, no task can preempt an already higher base priority task.

A higher priority task may require more than one resource already locked by lower priority tasks. However, due to PCP rules, only one task could have locked such resource on its host processor and increase its priority preventing the higher priority task to execute (arrival blocking). The other tasks only could have locked resources required by the higher priority task while migrated. As tasks are dispatched on their host processor by their leaving priority, no further arrival blocking is possible due to lower priority tasks. □.

Lemma 4. Nested MrsP does not suffer from deadlocks.

Proof. The source of deadlock in nested resources systems is when two or more resources requiring each other prevents any progress to be made. By Rule 1 forcing irreflexive partial order, circular dependencies and thus deadlock due to them are avoided. □.

Lemma 5. A safe upper bound to the number of concurrent access attempts to a resource r^j is given by $|V(r^j)| + |map(G(r^j))|$.

Proof. The number of direct accesses is safely bound by $|map(G(r^j))|$ as all direct accesses from tasks are outermost accesses, and thus are all dealt while not being help (and migrated), so this directly inherits all PCP properties. As only one request can be generated at a time from each processor, there is an upper bound on the number of processors from where the resource can be accessed. As shared resources have mutual exclusion, only one task can be requesting its inner resource at a time. The number of concurrent requests from outer resources is thus bounded to the number of such resources, i.e. $|V(r^j)|$. □.

Lemma 6. The cost of each individual access (e') to a resource r^j is bounded by $e'^j = c^j + \sum_{r_k \in \mathbf{U}(r_j)} n_j^k e^k$.

Proof. As a consequence of Rule 1, there is at least one terminal resource r^t in the system not accessing any inner resource, i.e. $\mathbf{U}(r^t) = \emptyset$. For such a resource, its individual access cost is:

$$e'^t = c^t$$

From Lemma 5, if we simplify the queue of a resource as $q^j = |V(r^j)| + |map(G(r^j))|$ then the total access cost to e^t is:

$$e^t = q^t c^t \Rightarrow e^t = q^t e'^t$$

For the set of resources accessing the terminal resource, $\mathbf{V}(r^t)$, the individual access cost can be expressed as the execution time of the resource plus the access cost to its inner resource r^t as:

$$e'^{t+1} = c^{t+1} + n^t_{t+1}(q^t * c^t)$$

then substituting e^t:

$$e'^{t+1} = c^{t+1} + n^t_{t+1}e^t \Rightarrow e^{t+1} = q^{t+1}(c^{t+1} + n^t_{t+1}e^t)$$

By the common method of recursion proving, it can be demonstrated[4] that this recursion holds for an arbitrary level k of nesting, where:

$$e'^k = c^k + n^{k-1}_k e^{k-1}$$

This can be directly applied to resources sequentially requiring more than one different independent inner resources:

$$e'^k = c^k + \sum_{r^{k-1} \in \mathbf{U}(r^k)} n^{k-1}_k e^{k-1}$$

Theorem 1. Equation 5 is a safe upper bound to the cost of accessing a MrsP shared resource and its required inner resources.

Proof. By construction if Lemma 5 gives a safe upper bound on the number of possible concurrent accesses to a resource r^j and Lemma 6 reflects a safe upper bound on the cost of each individual access to r^j and its required inner resources, then equation 5 is a safe upper bound to the cost of accessing r^j. □.

6 Conclusions

MrsP is a resource control protocol providing a safe upper bound on the contention for global shared resources on multiprocessor systems. In this paper we have provided a detailed approach to nested resource access under the MrsP protocol. By implementing local PCP control on each processor, the number of concurrent accesses to a global resource is bounded to at most one per processor on systems not considering nested resource access. By defining a helping mechanism by which busy waiting tasks can undertake progress inside shared resources on behalf of locally preempted tasks, the total access cost to a resource is effectively bounded. Based on this global resource control scheme, the PCP Response Time Analysis can be used to analyse MrsP systems incorporating its specific resource access cost analysis.

The approach presented in this paper defines a complete fine grained approach to nested resources for MrsP systems. The potential shortcomings of the helping mechanism when used in nested resources systems are addressed specifically. In particular, we have clarified under which circumstances tasks are eligible

[4] The complete proof can be found at `http://www.dit.upm.es/~jgarrido/mrsp/ae17-appendix.pdf`

to help and to be helped. We have also defined how active priorities are updated under our MrsP nested resources approach. Future work will consider analysis that uses more detailed knowledge about resource usage to reduce the pessimism within the analysis presented in this paper.

References

1. Biondi, A., Brandenburg, B.B., Wieder, A.: A blocking bound for nested FIFO spin locks pp. 291–302 (2016)
2. Block, A., Leontyev, H., Brandenburg, B.B., Anderson, J.H.: A flexible real-time locking protocol for multiprocessors. In: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 47–56. IEEE (2007)
3. Burns, A., Wellings, A.: Locking policies for multiprocessor ada. ACM SIGAda Ada Letters 33(2), 59–65 (2013)
4. Burns, A., Wellings, A.J.: A schedulability compatible multiprocessor resource sharing protocol – MrsP. In: Real-Time Systems (ECRTS), 25th Euromicro Conference on. pp. 282–291. IEEE (2013)
5. Calandrino, J.M., Leontyev, H., Block, A., Devi, U.C., Anderson, J.H.: Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In: Real-Time Systems Symposium. RTSS. 27th IEEE International. pp. 111–126. IEEE (2006)
6. Catellani, S., Bonato, L., Huber, S., Mezzetti, E.: Challenges in the implementation of MrsP. In: Ada-Europe International Conference on Reliable Software Technologies. pp. 179–195. Springer (2015)
7. Faggioli, D., Lipari, G., Cucinotta, T.: The multiprocessor bandwidth inheritance protocol. In: Real-Time Systems (ECRTS), 22nd Euromicro Conference on. pp. 90–99. IEEE (2010)
8. Lipari, G., Lamastra, G., Abeni, L.: Task synchronization in reservation-based real-time systems. IEEE Transactions on Computers 53(12), 1591–1601 (2004)
9. RTEMS, C.: Users guide-edition 4.6. 5, for rtems 4.6. 5. On-Line Applications Research Corporation (OAR)-http://www. 1tems. com 30 (2003)
10. Takada, H., Sakamura, K.: Real-time scalability of nested spin locks. In: Real-Time Computing Systems and Applications. Proceedings., Second International Workshop on. pp. 160–167. IEEE (1995)
11. Takada, H., Sakamura, K.: A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In: Real-Time Systems Symposium. Proceedings., The 18th IEEE. pp. 134–143. IEEE (1997)
12. Ward, B.C., Anderson, J.H.: Supporting nested locking in multiprocessor real-time systems. In: 24th Euromicro Conference on Real-Time Systems. pp. 223–232. IEEE (2012)
13. Ward, B.C., Anderson, J.H.: Multi-resource real-time reader/writer locks for multiprocessors. In: Parallel and Distributed Processing Symposium, IEEE 28th International. pp. 177–186. IEEE (2014)