2013 25th Euromicro Conference on Real-Time Systems

A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP

A. Burns and A.J. Wellings
Department of Computer Science,University of York,York, YO10 5GH, UK

***Abstract*—Lock-based resource sharing protocols for single processor systems are well understood and supported in programming languages and in Real-Time Operating Systems. In contrast, multiprocessor resource sharing protocols are less well developed with no agreed best practice. In this paper we propose a new multiprocessor variant of a protocol based on the single processor priority ceiling protocol. The distinctive nature of the new protocol is that tasks waiting to gain access to a resource must service the resource on behalf of other tasks that are waiting for the same resource (but have been preempted). The form of the protocol is motivated by the desire to link the protocol with effective schedulability analysis. The protocol is general purpose, but is developed in this paper for partitioned fixed priority systems with the sporadic task model. Two methods of supporting the protocol are described, as is a prototype 'proof of concept' implementation for one of these schemes.**

I. Introduction

Multiprocessor platforms are becoming more common and are currently the subject of much scrutiny from the real-time academic community [9]. Although there has been some success in determining the necessary support for scheduling, the issue of how best to support multiprocessor lock-based resource control protocols is far from clear. New results are emerging; indeed, there is a plethora of proposed schemes (see Section II); a consensus is, however, far from being realised.

In this paper, we take a somewhat different stance on the seemingly insurmountable problem of resource control on multiprocessor platforms. We first develop a schedulability-focussed viewpoint to motivate what properties *should* be provided by a resource sharing protocol for multiprocessor systems. An abstract architecturally neutral protocol is then defined that meets these requirements. Two potential implementation schemes for the protocol are then described. For one of these, a prototype implementation on a 4 core platform has been developed and is outlined in the final main section of the paper.

The paper is structured to follow the above argument. Initially, however, a review of existing protocols is provided (this is adapted from [13]), and then in Section III the system and task models are defined. Following the main sections of the paper, conclusions are provided in Section IX.

II. Multiprocessor Resource Control Protocols

Resource control policies for single processor systems are well understood. In particular:

Priority Inheritance Protocol (PIP) [18] – defined for pre-emptive fixed priority-based (FP) systems but also can be used with any system with static task-level measures of execution eligibility.

Priority Ceiling Protocol (PCP) [18] – defined for FP systems but also can be used with any system with static task-level measures of execution eligibility.

Non-preemptive Critical Sections (NCS) – can be used with any dispatching policy.

Stack Resource Policy (SRP) [3], [2] – defined as an extension to PCP for EDF systems but can be used with any static job-level execution eligibility. The SRP when applied to FP systems is sometimes called Priority Ceiling Emulation (PCE). Here we will use the acronym *PCP/SRP*.

Our approach is an extension of PCP/SRP and hence a short review of this scheme will be given in Section IV. Here we review the resource control protocols that have been defined for multiprocessors systems and extract their main characteristics.

With lock-based resource control protocols, locks can be *suspension-based* or *spin-based*. With suspension-based locking, if the calling task cannot acquire the lock, it is placed in (usually) a priority-ordered queue and waits for the lock. To bound the blocking time, priority inversion avoidance algorithms are typically used. With spin-based locking, if the calling task cannot acquire the lock, it busy-waits for the lock to become available. To bound the blocking time, the thread often spins non-preemptively (i.e, at the highest priority) and is placed in a FIFO or priority-ordered queue. The lock-owner may also run non-preemptively.

A. MPCP and DPCP

There have been several versions of the Multiprocessor Priority Ceiling Protocol (MPCP) that extend the single processor PCP. Tasks are fully partitioned amongst processors. Initially, Rajkumar et al.'s [16] proposed that all global resources were assigned to a single synchronization processor. This was then generalized in the same paper to allow multiple synchronization processors, but again each resource was assigned to one synchronization processor[1]. A task that wishes to access a global resource migrates to the synchronization processor for the duration of its access. With the basic MPCP, there was a single synchronization processor and nested resource accesses are allowed. The protocol ensured deadlock free access. With the generalized MPCP, nested resources are not allowed.

[1]The work by Lozi et al. [14] essentially adopts the same approach to help improve performance when legacy code is moved to a multicore architecture.

1068-3070/13 $26.00 © 2013 IEEE
DOI 10.1109/ECRTS.2013.37

Later, Rajkumar et al. [17], [15] renamed the above protocol as the Distributed Priority Ceiling Protocol (DPCP) and clarified the remote access mechanism. Hence, the protocol was targeted at distributed shared memory systems. For the globally shared memory systems, the need for remote access is removed and all global resources can be accessed from all processors.

B. MSRP

The Stack Resource Policy, proposed by Baker [3], emphasizes that: "*a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling.*" If allowed to lock any resource, the current executing job must be running at the ceiling of the resource. Once a jobs starts, all its resources will be available. With this approach, all the tasks in the system can use the same run-time stack (hence the name of the policy).

In order to preserve this property, Gai and Lipari [11] proposed the Multiprocessor Stack Resource Policy (MSRP) for partitioned EDF scheduling. Resources are divided into two groups: local and global. Local resources are only accessed by tasks which execute on the same processor. Global resources are those which can be accessed by tasks running on different processors. Unlike SRP, global resources have different ceilings: one for each processor. Furthermore, each processor has its own *system ceiling*. On processor k, tasks are only allowed to execute global resources at the processor ceiling priority, which is the highest preemption level of all the tasks on processor k.

Global resources are shared across processors in a FIFO manner. In order to acquire a global resource, a task must be running at the processor ceiling which makes it non-preemptive. In order to maintain the ability for all tasks on a processor to share the same stack, it is necessary to non-preemptively busy-wait for the resource.

C. FMLP

The Flexible Multiprocessor Locking Protocol (FMLP) proposed by Block et al. [4] supports both global and partitioned systems. Resources under FMLP are divided by the programmer into long and short resources. The protocol introduces the notion of a Resource Group, which can contain either short or long resources but not both. A group containing short resources is protected by a non-preemptive FIFO queue lock. A group containing long resources is protected by a semaphore lock. Groups contain resources that are nested, so that a task can hold more than one resource at a time. Non nestable resources are grouped individually. Group locks have the unfortunate side effect of reducing parallelism and are an impediment to composability.

D. OMLP

The $O(m)$ locking protocol (OMLP) is a suspension-based resource sharing protocol proposed by Brandenburg et al. [5]. The algorithm is proposed to reduce the interference from further priority inversion in FMLP caused by suspended tasks being blocked by long resources. Resources are either global or local. OMLP can be used in either global or partitioned systems. Here, we consider only partitioned systems.

Partitioned OMLP uses contention tokens to control access to global resources. There is one token per processor, which is used by all tasks on that processor when they wish to access a global resource. Associated with each token there is a priority queue PQ_m. There is only one queue per global resource; a FIFO queue FQ_k, again of maximum length m. In order to acquire a global resource, the local token must be acquired first. If the token is not free, the requesting task is enqueued in PQ_m. If free, the token is acquired, its priority is then raised to the highest priority on that processor, and the task is added to the global FQ_k and, if necessary, suspended. When the head of FQ_k finishes with the resource, it is removed from the queue, releases its contention token, and the next element in the queue (if any) is granted the resource.

Recently, Brandenburg and Anderson [6] have developed a new resource sharing scheme for clustered multiprocessor systems based on OMLP. The aim is to limit multiple repeated priority-inversion blocks that occur with a simple priority boosting scheme. Essentially, a high priority thread donates its priority to a lower priority thread to ensure that the lower priority thread is scheduled when holding a resource. Such a donation can, however, only be performed once.

E. SPEPP

This protocol, Spinning Processor Executes for Preempted Processor, was proposed by Takada and Sakamura in 1997 [19]. It is an example of a *helping* protocol; in which a task waiting to get a resource will execute the resource's critical section on behalf of another waiting task. Resources are accessed in FIFO order with the resource operations executed non-preemptively. However, the designers of SPEPP did not want the waiting tasks to also spin non-preemptively as this reduces schedulability for high priority (short deadline) tasks.

F. M-BWI

The Multiprocessor BandWidth Inheritance protocol (M-BWI) [10] is an extension of the single processor BWI. This is developed for soft or open real-time systems in which tasks are executed within execution-time *servers* that limit the amount of processor time each task can consume. To deal with the problem that a task while holding a resource may run out of budget [8] or is preempted, M-BWI allows the resource holding task to use the budget from other tasks that are waiting to gain access to the same resource. It does this by allowing the lock holding task to migrate between servers. A task waiting to access a resource is either busy-waiting (using its server's budget) or its server is being used by the migrated lock holding task. Access to the resource is constrained to be FIFO.

G. Fiasco-SMP

Fiasco-SMP is a port of the Fiasco microkernel for the multiprocessor-x86 architecture [12]. It shows how migrating critical sections can be achieved efficiently.

H. Summary

This brief review has indicated that there has been a wide variety of protocols proposed. Most are complex and may have significant run-time overheads. Others impose restrictions such as no nested resources.

The protocol developed in this paper is similar to MSRP but has one crucial difference: tasks that busy wait can use their 'spin' time to undertake computation on behalf of other waiting tasks. In this way the protocol follows the approach of SPEPP, but does not require non-preemptive execution of resource operations. It can also be viewed as an adaptation of the M-BWI or Fiasco-SMP.

The main contribution of this paper is a multiprocessor protocol having all the desirable properties of the single processor PCP/SRP (see Section V). We refer to the protocol by the acronym MrsP (Multiprocessor resource sharing Protocol); using some lower case letters to help distinguish it from similar acronyms (e.g. MSRP).

III. System and Task Model

Our aim is to develop a general purpose protocol that is applicable to partitioned, semi-partitioned and globally scheduled systems using fixed priorities, EDF or any other designation of urgency. In this paper, however, we restrict consideration to *fully partitioned systems* where tasks are scheduled using *fixed priorities*. We employ the general *sporadic task model*. Each task (τ_i) in the system is characterised by its period, T_i, deadline, D_i and worst-case computation time, C_i; it can give rise to a potentially unbounded sequence of jobs. Deadlines are *unconstrained*, but no two jobs from the same task can be active (executable) at the same time. The priority of task τ_i is $Pri(\tau_i)$ (Pri will also be used to give the ceiling priority of a resource).

The execution platform consists of m identical processors, numbered p_1 to p_m.

Resources (r) are shared between sets of tasks. They must be accessed under *mutual exclusion*. The code associated with a resource is termed a *critical section*. Tasks and resources are related by the two functions: $G(r^j)$ returns the set of tasks that use resource r^j, and $F(\tau_i)$ is the set of resources used by task τ_i. The worst-case execution time of resource r^j when accessed by task τ_i is in general denoted by c_i^j. However, for ease of presentation we assume that this value is not dependent on the actual task accessing the resource, and hence the parameter c^j is used throughout. None of the analysis presented is fundamentally altered by this simplification. It would be straightforward to define a set of interfaces for each resource and to have tasks call different interfaces; indeed a single task may call the same resource a number of times using different interfaces. Each interface could be given a distinctive worst-case execution time.

As a property of the static partitioning of the tasks, we have a function map that takes a set of tasks and returns the set of processors onto which the tasks have been allocated. We shall make use of the entity e^j which is an execution time parameter of resource r^j; it is defined by $e^j = |map(G(r^j))|c^j$ – where $||$ gives the size of the set.

IV. PCP/SRP for Single Processor Systems

All priority ceiling protocols, when applied to fixed priority single processor systems, assigns priorities to resources based on the usage of the resource, so $Pri(r^j) = \max_{\tau_k \in G(r^j)} Pri(\tau_k)$. At run-time, when a task accesses a resource, its priority is immediately raised to the ceiling priority of the resource. The protocol has the following properties:

- 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 will be available.
- Deadlocks are prevented.

Blocking is when a task is prevented from making progress due to another task with a lower base priority having a current higher active priority (as it is accessing a resource with a higher ceiling priority).

Scheduling analysis, in the form of Response-Time Analysis (RTA) [1] incorporates PCP/SRP via the following equation:

$$R_i \;=\; C_i \;+\; B_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

where $\mathbf{hp}(i)$ is the set of tasks with priority greater than τ_i, and B_i is the blocking term introduced by the priority ceiling protocol. In general, the blocking term is the maximum of a number of values derived from the application code and the implementation of that code on a RTOS:

$$B_i \;=\; \max\{\hat{c}, \hat{b}\} \qquad (2)$$

where $\hat{c}$ is the maximum computation time of all resources that are used by at least one task with priority less than that of τ_i and at least one task with an equal or higher priority; and $\hat{b}$ is the maximum non-preemptive execution time induced by the RTOS.

All tasks make use of the API provided by the RTOS. Although fine grain locking is usually provided by the RTOS, each task will use some non-preemptive code; for example the code used at job termination to place the task on a delay queue and to switch the processor to the next eligible task to execute. It follows from equation (2) that if $\hat{c} < \hat{b}$ (the worst-case blocking induced by the application is less than that induced by the RTOS) then there is no benefit in using any priority ceiling protocol. All resource accesses may as well be non-preemptive. The fact that PCP/SRP is supported in most RTOSs and real-time programming languages is testament to the observation that, in general, $\hat{c} > \hat{b}$. This observation remains valid with multiprocessor systems.

Note we can also decompose the C parameters by noting:

$$C_i = WCET_i + \sum_{r^j \in \mathbf{F}(\tau_i)} n_i c^j \qquad (3)$$

where $WCET_i$ is the worst-case execution time of the task, ignoring the time it takes executing whilst accessing resources (but including all time spend in the RTOS), and n_i is the number of times τ_i uses r^j.

V. Desirable Properties for MrsP

One of the consequences of executing a set of tasks on a single processor is that the serialisation of the task's execution (due to there being only a single processor) 'hides' the necessary serialisation over the use of other resources. As the number of processors is increased, the need to serialise the execution of the other resources is exposed. So, if there can be n parallel (rather than just concurrent) accesses to a resource r^j, it is reasonable to expect the time it takes to gain access to be $(n-1)c^j$ (i.e. a FIFO queue of tasks waiting their turn to use the resource). And hence the full cost of accessing the resource to be nc^j, ($n-1$ accesses from other processors plus the task's own execution). From a schedulability point of view, the move to a multiprocessor platform is a tradeoff between

- a reduction in interference from other tasks, and
- an increase in the time it takes to access shared resources.

So, for example, a move from one processor to two will half the contention for the processor but double the time it takes to access shared resources. For any realistic software system, the overall effect will be significantly positive.

The intuition underlying the proposed protocol is that the schedulability analysis for a partitioned multiprocessor should be identical to that of a single processor except that the cost of accessing each resource is increased to reflect the need to serialise access from potentially parallel access. In the system model (section III), the entity e was defined. For any resource, r^j, the number of processors on which tasks that use the resource can execute is given by $|map(G(r^j))|$; each access costs c^j so the full execution time cost for the resource is given by: $e^j = |map(G(r^j))|c^j$. Note the value of e^j is not directly a function of the number of tasks that use the resource, rather it just reflects the number of potential parallel accesses. Consequently, if a resource is only used by tasks executing on a single processor then $e^j = c^j$.

Under MrsP we incorporate the property, fundamental to the PCP/SRP protocol, that once a task starts executing, its resources will be logically available – but the execution time required to use the resource is e^j, not c^j.

It follows from these intuitions that the Response-Time Analysis for each task on a multiprocessor should take the following form:

$$R_i \;=\; C_i \;+\; \max\{\hat{e}, \hat{b}\} \;+\; \sum_{\tau_j \in \mathbf{hpl}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (4)$$

where $\mathbf{hpl}(i)$ is the set of *local* tasks with priority greater than τ_i, and $\hat{e}$ is the maximum execution time of a resource that is used by a local task with priority less that than of τ_i and a local task with an equal or high priority; $\hat{b}$ is the implementation induced blocking term used earlier – in equation (2).

The C parameter for each task is now:

$$C_i = WCET_i + \sum_{r^j \in \mathbf{F}(\tau_\mathbf{i})} n_i e^j \qquad (5)$$

In addition, to be compatible with single processor PCP/SRP, nested resource usage should be allowed and deadlock free execution ensured. We now develop a protocol that will deliver this scheduling analysis for its worst-case behaviour.

It is clear from the form of the schedulability equations (4 and 5) that they will not capture the worst-case behaviour of the system if suspension-based queueing for resources is employed. With suspension-based protocols, queues could be longer (as more than one task from the same processor can be on the queue) and extra blocking terms must be included – every time a task suspends it can be subject to priority inversion if a lower priority task executes and accesses a resource with a higher ceiling priority. *We conclude that some form of FIFO spinning must be used.* But at what priority should a task spin (and then when successful, use the resource)? Two protocols are discussed in the literature:

1) Spin non-preemptively, or
2) Spin at local ceiling priority.

The first approach is modeled by equations (4 and 5) if all resources are deemed to be used by the highest priority task on each processor. But the result could be poor schedulability; the highest priority task (with the shortest deadline) on each processor must be able to cope with the largest blocking term on its processor. On a single processor, non-preemptive resource usage is not acceptable, rather some form of priority ceiling is used. The same argument applies to multiprocessors, non-preemptive execution is acceptable if the resulting system is schedulable but it is not an adequate general-purpose protocol.

The second approach can, however, result in prolonged blocking as the task with the resource lock could be locally preempted leading to long delays for the other waiting tasks on different processors. It, therefore, does not lead to the worst-case behaviour captured by equations (4 and 5). But spinning only at the local ceiling does have the advantage of not inducing excessive blocking on the higher priority tasks. *The development of MrsP is motivated by the wish to spin only at the local ceiling level, but to have bounded blocking leading to the applicability of equations (4 and 5).*

VI. Definition of MrsP

We start with non-nested resources. Of the existing protocols, the one that most closely matches our requirements is the Multiprocessor Stack Resource Policy (MSRP) (see Section II-B). However, it does not lead to the analysis outlined above – as the time it takes to gain access to a resource can be extended significantly by preempting tasks. Here we define a variant of MSRP (which we term MrsP). The basic aspects of the protocol are:

1. All resources are assigned a set of ceiling priorities, one per processor (for those processors that have tasks that use the

resource); for processor p_k it is the maximum priority of all tasks allocated to p_k that use the resource.

2. An access request on any resource results in the priority of the task being immediately raised to the local ceiling for the resource.

3. Accesses to a resource are dealt with in a FIFO order.

4. While waiting to gain access to the resource, and while actually using the resource, the task continues to be active and executes (possible spinning) with priority equal to the local ceiling of the resource.

The first point is represented formally by:

$$Pri(r^j, p_k) = \max_{\tau_i : \tau_i \in G(r^j) \ and \ map(\tau_i) = \{p_k\}} \{Pri(\tau_i)\}$$

where $Pri(r^j, p_k)$ is the ceiling priority of resource r^j on processor p_k.

The setting of the local ceiling for the resource and the raising of the accessing task's priority to the ceiling level means that each processor implements a local PCP/SRP protocol. As a result, the following properties hold (there are expressed as Lemmas).

Lemma 1: At most one task per processor can be attempting to access any specific resource.

Proof Follows directly from the properties of PCP/SRP; if a task is attempting to access an arbitrary resource r^k it remains active and can only be preempted by a local task if that task has a higher priority than the resource's ceiling. Such a task will not access the resource and so at most one task per processor can be accessing r^k □.

Lemma 2: The maximum length of the FIFO queue for resource r^k is $|map(G(r^k))|$.

Proof Follows directly from the previous Lemma; only one task per involved processor can be accessing or using the resource □.

Note that if the task only spins at its own priority, or a suspension scheme is used then the maximum length of the queue could be as high as $|G(r^k)|$.

These properties of the protocol are not particulary novel. However MrsP requires a further feature if its worst-case behaviour is to match the proposed schedulability analysis. To motivate this feature, first consider the behaviour of the protocol if *no* requesting task is locally preempted. In this situation it is clear that the maximum time it takes for some task τ_i to progress through the FIFO queue associated with resource r^k is $|map(G(r^k))|$ times c^k (including τ_i's usage). But if any participating task is locally preempted while it should be using the resource then this time interval will be increased. And indeed it could be increased by a significant amount as it must include the total executing time of the preempting job. For this reason, some protocols explicitly require that resource accesses must be non-preemptive. In effect this implies that all resources are 'logically' used by the highest priority task on each processor. As noted above, the resulting system may be schedulable, but in general it gives a sub-optimal solution. The highest priority tasks (i.e., the ones with the shortest deadlines) must be capable of dealing with the longest blocking times in the system.

For MrsP, we take another approach; one employing the idea of helping used in SPEPP, M-BWI and Fiasco-SMP – see Section II. The protocol is completed by the following:

5. Any task waiting to gain access to a resource must be capable of undertaking the associated computation on behalf of any other waiting task.

6. This cooperating task must undertake the outstanding requests in the original FIFO order.

Hence, for example, if a resource (r^k) is accessed by two tasks (τ_1, τ_2) from two distinct processors then the critical sections associated with the two requests can be executed by any of the two tasks. So if τ_1 accesses the resource first, but is locally preempted whilst it has the lock on the resource, then if τ_2 accesses r^k it will first 'take over' the execution for τ_1, complete the critical section for τ_1, and it will then execute the critical code on its own behalf. When τ_1 executes again it will find its access to r^k has been completed.

The key notion here is that a task is not spinning uselessly while it is waiting to access the resource, rather it is prepared to use its 'wasted cycles' to help other tasks make progress. This is a significant property. We postpone until the next section how this property can be implemented; first we show that it gives the desirable worst-case behaviour.

Lemma 3: Each job can suffer at most a single local block, and this blocking will occur before it actually executes.

Proof As a task will execute with the local resource ceiling while waiting or accessing a resource, MrsP will behave according to the behaviour of PCP/SRP. In particular, once a job starts executing, no newly released job can preempt it and then access any resource it might need. Moreover, the job can only start executing if it has the higher priority, and hence again no resource it may access can be locked (or indeed accessed) at that time by a local task. □.

Theorem 1: The worst-case response time of a task executing under MrsP is given by equations (4) and (5).

Proof Consider general task τ_i. Every time it accesses a resource, in the worst-case, it will have to undertake the execution for the resource on behalf of the maximum number of requests in the FIFO queue. Even if τ_i is locally preempted while waiting to gain access to the resource, when it next executes it will still have (at most) the same work to do (in the worst case). It follows that equation (5) captures the worst-case execution time for the task.

The maximum blocking time induced by the application code (as apposed to the RTOS) is the maximum time a lower priority task will execute with an inherited ceiling priority equal or higher than the priority of τ_i. This is exactly the value used in equation (4) □.

Note that one of the RTOS actions that may contribute to the RTOS induced blocking term $\hat{b}$ is the operation needed to place a task at the back of the FIFO spinning queue.

Given that the protocol can be implemented efficiently we have a scheme that is, intuitively, the same as PCP/SRP – the cost for accessing a resource has increased but the response-time framework that is 'standard' for the analysis of single processor systems can be applied unchanged (i.e., equations

(4) and (5) are of the same structure as equations (1) and (3)). As argued earlier, although the cost associated with the resource has increased, this is heavily countered by the reduction in contention for the processor(s). For partitioned systems, there are a number of factors to consider when deciding upon the scheme to use for generating the task to processor mapping. Although beyond the scope of this paper, it is clear that mappings that reduce the amount of parallel contention for resources could be a useful metric to apply.

Finally, in this section, we note that the proposed scheme dominates those schemes that involve the accessing task spinning at the local ceiling level. MrsP has the same behaviour if there is no local preemption, and reduces waiting time if there is preemption. It, therefore, always performs better for the tasks actually involved in resource usage. Note also that the use of a local ceiling is the minimum ceiling value to use if the benefits of PCP/SRP are to be gained.

A comparison with a non-preemptive protocol inevitably involves a trade-off between the impact on high and low priority tasks. If a low priority task accesses a resource non-preemptively then clearly the reaction time for the low priority task is minimised. But any higher priority task suffers blocking even when it is not contending for the resource. Even on a single processor systems, if all tasks are schedulable when all resources are accessed non-preemptively then this is a sensible straightforward protocol to employ. The whole point of any resource access protocol is, however, to constrain priority inversion to where it is actually necessary; and thereby increase the likelihood of obtaining a schedulable system. MrsP will, therefore, always outperform a non-preemption policy in the sense that it will have less (or at worst equal) impact on higher priority tasks.

A. Nested Resource Usage

Now we move to nested resource usage (as supported in [20]) and the issue of deadlock-free execution. These are crucially important properties of PCP/SRP. For nested resources, MrsP follows the same behaviour as PCP/SRP, resulting in extra time spent executing with the resource. We shall use an example to motivate the analysis for nested resources usage. Consider a system with four tasks, τ_1, ..., τ_4, executing on four different processors, and two resources, r^1 and r^2, with execution times c^1 and c^2. 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. The direct execution time for τ_1 will include $c^1 + c^2$.

Now consider the extended execution times required by MrsP. Each access of r^1 will, in the worst-case involve waiting for first τ_3 and then τ_4 to complete their usage of r^2. Hence for τ_1 and τ_2 their call on r^1 will cost $2(c^1 + 3c^2)$. The outer '2' as two processors are involved with r^1, and the inner '3' as there could be two processors ahead of the caller in accessing r^2. For τ_3 and τ_4, their accesses of r^2 would cost in the worst case $3c^2$. Although there are four processors with tasks that access r^2, only one of τ_1 and τ_2 could be queued on r^2, as their accesses are serialised through r^1.

Generalising from this example, we can derive the required formulas for the time it takes each task to access a, potentially nested, resource. First, we note that if one resource (r^a) accesses another (r^b) then at most one task from the set of tasks that access r^a can be waiting to access r^b. As noted above, r^a serialises the accesses to r^b as only one task can be executing within r^a. Define $V(r^i)$ to be the set of resources that access r^i; hence $|V(r^i)|$ is the number of such resources. Each of these can give rise to a queuing task (on r^i). Tasks can also, of course, call r^i directly (as in the non-nested case). This means that the maximum size of the FIFO queue of task waiting to access r^i (including the task actually using the resource) is $|V(r^i)| + |map(G(r^j))|$. Each entry can give rise to c^i computation time and hence the total time spent accessing the resource is given by:

$$e^i \;=\; (|V(r^i)| + |map(G(r^i))|)c^i \tag{6}$$

This is a safe upper bound. For a particular set of tasks and resources, and allocations to processors, it may be possible to reduce the size of this term.

Returning to the example, we noted that the cost of access to r^1 for τ_1 and τ_2 is $2c^1 + 6c^2$, and to r^2 for τ_3 and τ_4 is $3c^2$. Now consider using a single group lock for both resources (as a number of protocols do in order to prevent deadlock). For τ_1 and τ_2 the cost is now just $2c^1 + 2c^2$; this comes from the worst-case FIFO queue for the single group lock: τ_1 and τ_2 are accessing r^1, and τ_3 and τ_4 are accessing r^2. So the cost of access is reduced. But for τ_3 and τ_4, the cost of their use of r^2 is now increased from $3c^2$ to $2(c^1 + c^2) + 2c^2$ because there are four tasks (on four processors) accessing the group lock. Hence there is a tradeoff between the use of nested or only group locks. However, this is schedulability issue, group locks are not required in MrsP for deadlock prevention.

On a single processor, the correct assignment of priorities serialises access to resources so that circular chains of hold/request accesses to resources do not arise at run-time, and hence deadlocks cannot occur. On a multiprocessor, this constraint must be ensured by extending the protocol. A common deadlock prevention scheme is to statically order resources and to only allow access to resources with an order number greater than that of any currently held resource. Although this may be considered a restriction on the expressive power of the computational model (and open to issues with legacy code), it is more expressive than those protocols that ban nested usage or require a group lock – that has the same impact as non-nesting. In a programming domain the use of exceptions for 'out of order' requests allow robust programs to be developed. We have investigated the use of nested but ordered locks for general multiprocessor resource control protocols elsewhere [13] (though not for MrsP). The adoption of ordered locking for MrsP is straightforward, though not covered further in this paper.

VII. Realising MrsP

In this section we describe two possible approaches for implementing MrsP. The first uses thread migration, the second uses parallel execution. The key to both schemes is that a processor that has a busy-waiting task can undertake work for any (or indeed all) of the other tasks waiting to gain access to the same resource.

A. A Task Migration Approach

Although the scheme developed in this paper is focussed on partitioned systems, the ability for a task to migrate from one processor to another at run-time is a common property of multiprocessor operating systems. This provides a straightforward means of implementing MrsP. If a task, τ_a, is preempted whilst accessing a resource, r^i, (by a higher priority local task) then τ_a can migrate to any processor on which a task is spinning waiting to gain access to the same resource. On this new processor τ_a is given the priority one higher than the spinning task so that it preempts the spinning task. This simple scheme is sufficient but does require that the priority values one above resource ceilings is unused (i.e. no tasks or resources are given this priority value). But as a consequence, the requirement of the protocol: *that any spinning task can take over the resource access on behalf of any other task* is satisfied. The spinning task just gives way to the migrating task as the new task has a slightly higher priority[2]. In practice, the supporting RTOS must be aware that there is a separate priority associated with each processor in the thread's affinity set.

The act of migrating a task has a run-time overhead that must be taken into account when evaluation this means of implementing MrsP. The cost of two migrations must be less than the added interference from the high priority tasks that are delaying the resource requesting task (perhaps on another processors). Also a migrating task is likely to suffer from an execution time penalty as its local cache can not be utilised. However, a preempted task is also likely to have its data in cache overwritten by the time it executes again.

To give an example of the migration approach. Assume r^1 is used by tasks on three processors: p_1, p_2 and p_3. At some time t, a task (τ_a) on p_1 executes and gains access to r^1 in doing so its priority is raised to $Pri(r^1, p_1)$. Slightly later, task τ_q on processor p_2 attempts to gain access to the resource. It first has its priority raised to $Pri(r^1, p_2)$ and it then spins at this priority. Then τ_a is locally preempted by τ_h as this task has just been released and $Pri(\tau_h) > Pri(r^1, p_1)$. A migration then takes place; τ_a could potentially move to p_2 or p_3, but there is only a spinning task on p_2. Hence, τ_a migrates to p_2 where it will execute with priority $Pri(r^1, p_2)+1$. It will stay on this processor unless it is again preempted, it which case it may move back to p_1 (or to p_3 if a request has been made from that processor). When τ_a has completed its use of the resource it will migrate back to its original processor (p_1) to continue it normal execution at its base priority. At that time τ_q will gain access to the resource and execute on processor p_2.

A prototype implementation of a scheme based on migration is included in Section VIII. It uses affinities to control the migrations necessary.

[2]For protocols that are based on suspension, Brandenburg and Bastoni [7] have suggested that priority inheritance algorithms should include processor affinity mask inheritance as well. This effectively allows a task holding a resource to migrate.

B. A Duplicated Execution Approach

A more speculative implementation approach is possible if resource critical sections have the property of also being *atomic actions*, in the sense that the resource has internal state (S) and the code of the resource takes 'input' and state and produced output and updated state with no side effects: $(input, S) \Rightarrow (output, S')$. This is the assumption used in the SPEPP protocol (see section II-E).

With this structure it is possible for all 'waiting' tasks to actually execute the code for each request, and to do that in parallel with other resource users (on other processors). The first to finish actually updates S and produces the output. Later finishers have a null commit on the resource.

VIII. Prototype Implementation of MrsP

In this section we describe a proof of concept implementation of the first of the two schemes outlined in the previous section. We focus on an actual implementation rather than a simulation (as many other paper do, for example M-BWI) as the theoretical properties of MrsP are clear. What is important is to demonstrate that the scheme is a viable one when real overheads are encountered.

The basis of the implementation is to dynamically associate a set of affinities with each resource. An attempt to lock a resource adds a processor to the resource's affinities. While accessing the resource, the task inherits the resource's affinities and hence it can execute on any processor that has a waiting (spinning) task. The FIFO queue is implemented as an array of spin locks (one per resource). The following pseudo code outlines the implementation; in this code R is the resource, t is the accessing task and p is the processor from which the lock request is being made (i.e. t is executing on p). The code also keeps track, in R(t), of the id of the task that currently holds the lock (if there is one, otherwise R(t) is null). When the request to lock R is made, the protocol raises the priority of t to the local ceiling and sets the affinity of the resource to include its own affinity. If the resource is already locked, it sets the current lock-holder's affinity to be that of the resource and spins at the next available position in the FIFO queue. The task continues to spin until it is released (by the action of some other task calling `Unlock`).

Note this pseudo code does not contain the additional synchronisations required to ensure that certain sequences of actions are atomic. The actual code uses a simple spinlock per resource to achieve this atomicity.

```
Lock (R, t, p) ->
  raise priority of t to local ceiling of R
  Affinities(R) := Affinities(R) + p
  if already locked
```

```
    get current resource user R(t)
    Affinities(R(t)) := Affinities(R)
    obtain FIFO lock on R and spin
  else
    Affinities(t) := Affinities(R)
  end if
  set current lock holder to self
  raise priority of t by 1
  -- use R

Unlock(R, t, p) ->
  Affinities(R) := Affinities(R) - p
  Release next task in FIFO queue (if there is one)
  Affinities(t) := p
  lower priority of t to its base value
```

It should be noted that all code is run at the ceiling priority of the resource, and hence the protocol does not contain any non-preemptive sections.

A. Experimental Results

Our experiments are performed on an Intel(R) Core(TM) Quad CPU i7-3770 running at 3.40GHz. Each core supports hyper-threading, but this is turned off for our experiments. The four cores are numbered 0, 1, 2 and 3. We run Linux kernel 3.5.4, which supports real-time priorities, affinities, preemption and FIFO scheduling within priority. We use the Ada 2005 compiler from AdaCore and set affinities of threads by dropping into C and calling the Linux API directly. Ada handles the priority settings and changes.

The Linux OS ensures that if a task is locally preempted it will migrate to any other core in its affinity set if it would be the highest priority task on that core. It was not possible on the experimental test-bed to give this task a different priority on each core, and hence ceilings were chosen so that, for each resource, the local ceiling was the same on each core. In future work, we will modify the Linux kernel to provide more direct support for the MrsP protocol.

A number of experiments are performed to demonstrate that the scheme is implementable, to show that it can outperform other protocols and to give some indication of the overheads likely with the scheme.

Throughout this section, we consider three protocols all based on spinning when the resource is already allocated. The first is the protocol developed in this paper. The second is a simple non-preemption protocol where the accessing task raises its priority to the maximum value and then spins on a FIFO queue to gain access. The third protocol again uses a FIFO queue to access to the resource but the client task now spins at the local resource ceiling. With this single resource experiment, this third protocol behaves in an identical way to MrsP except that there is no migration and is identical to the non-preemption protocol except in the priority used. The use of non-preemption and 'spinning at ceiling' cover the two most distinctive features of the existing protocols (see Section II).

Each experiment is performed 500 times and the results are shown at the 95% confidence level.

Basic Protocol Overheads

MrsP: For Lock: a priority change to the ceiling; a spinlock to gain access to the resource's data structure; the setting of two affinity masks (one for the locking holding task, the other for the resource); spinning in a FIFO queue if the resource is locked. For Unlock: a spinlock to gain access to the resource's data structure; releasing any queued task; the setting of two affinity masks (one for the locking holding task, the other for the resource); priority change to base priority.

Non-preemption: For Lock: a priority change to execute non-preemptively; a spinlock to gain access to the resource's data structure; spinning in a FIFO queue if the resource is locked. For Unlock: a spinlock to gain access to the resource's data structure; releasing any queued task; priority change to base priority.

Ceiling: For Lock: a priority change to the ceiling; a spinlock to gain access to the resource's data structure; spinning in a FIFO queue if the resource is locked. For Unlock: a spinlock to gain access to the resource's data structure; releasing any queued task; priority change to base priority.

In our experiments, all the protocols are implemented at the application-level using the basic Linux API for setting affinity and setting priority. Hence, each of these operations requires a system call. If the protocols were integrated into, say, the Linux mutex access protocols then there would be a significant reduction in the number of system calls. The key differences between the run-time costs of the protocols is that MrsP will result in tasks migrating. On our system, the typical cost of a task being migrated is around 8 microseconds.

The first experiment is designed to show the overall properties of MrsP and contrast its performance with the non-preemption and local ceiling approaches. The experimental setup is that on core 1 we have a high priority task that requires no resource access, and a low priority task that shares a resource with another low priority task on core 3. The experiment is run on an otherwise unloaded system; any interrupt handling for general Linux is performed on core 0. Only interrupts to support scheduling occur on cores 1 and 3.

The high priority task is effectively released immediately after the low priority task (that is allocated to its machine) has acquired the resource. It simply performs some computation in a loop, as shown below. The number of cycles of the loop can be set to determine the amount of computation performed. Each cycle takes approximately 25.7ns.

```
for i in 1 .. Number_Of_Cycles_In_HP loop
  -- a series of calculations
end loop;
```

The low priority tasks are identical on cores 1 and 3. They are released at the same time. However, the one on core 1 is pre-allocated the lock. Once released (and the lock has been obtained if not pre-allocated), they perform some computation, unlock the resource and terminate. The computation performed with the lock is structured similarly to the above (although the the execution time for each cycle is significant less at 0.47ns).

In Figures 1 to 3, we show the response times of the low priority task on core 3 and the low priority task on core 1. All figures show the response times for the three protocols in the order (left to right): Non-preemptive, MrsP, Ceiling. Note the response time of the low priority task is considered to be when

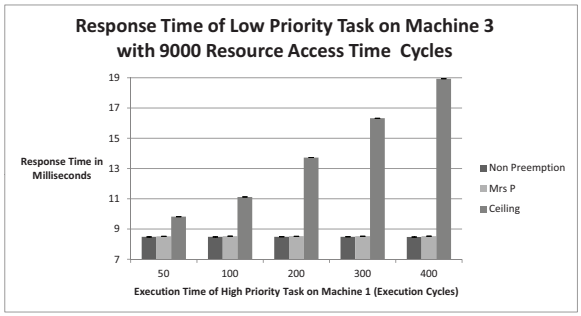

Fig. 1. Response Time of the Low Priority Task

it finishes accessing the resourse. Of course, it would then need to migrate back to its original core to fully terminate. We are concerned with trends here rather than absolute execution times.

For the low priority task, our expectation is that the non-preemption algorithm should outperform the others; as, in the worst case, the LP task on core 3 has only to wait for the other LP task to finish with the resource before it can acquire the resource and finish. With the simple FIFO ceiling spinning protocol, we expect poor results. The LP task on core 3 must also wait for the HP task to finish (as it will preempt the other LP task). We expect MrsP to outperform the FIFO ceiling spinning algorithm, as the resource holding LP task can migrate to core 3 when it is preempted by the HP tasks.

The results in Figure 1 confirm this. The resource access time in this experiment is high (9000 resource cycles) and we consider the response time against varying high priority execution times. As expected, when using the ceiling algorithm, the response time of the low priority task is dominated by the execution time of the high priority. Hence, the higher this becomes, the worst the response times are. These experiments show little difference between the MrsP and non-preemption protocols.

For the high priority task, in contrast to the previous experiment, we now expect the non-preemption algorithm to perform badly as the HP task cannot execute when released as the LP task is non-preemptive. We expect the FIFO ceiling spinning and MrsP algorithms to be approximately the same as each HP task can preempt the resource holding LP task. Again this is confirmed by the results in Figure 2. Here we consider a high priority task with only 1500 execution cycles and vary the resource access time. Now there is a noticeable difference between the performance of MrsP and the Ceiling protocol due to the overheads of the protocol. As the resource access time decreases, eventually the non-preemption algorithm performs better than MrsP, as illustrated in Figure 3.

In our final experiment we consider the impact of two migrations. Here we consider three cores. Identical high priority tasks run on cores 1 and 2, and identical low priority tasks run on cores 1, 2 and 3. The low priority tasks share a single resource. The execution of the three low priority tasks are contrived so that they request the resource in the order low

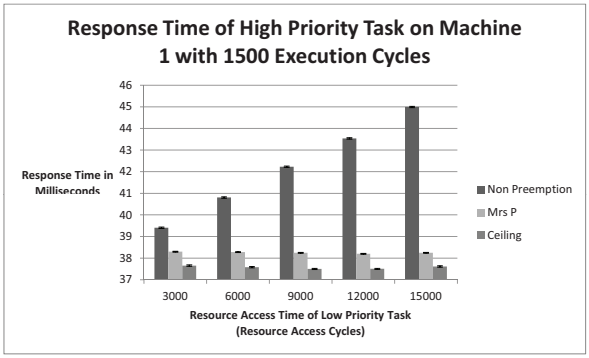

Fig. 2. Response Time of the High Priority Task (1)

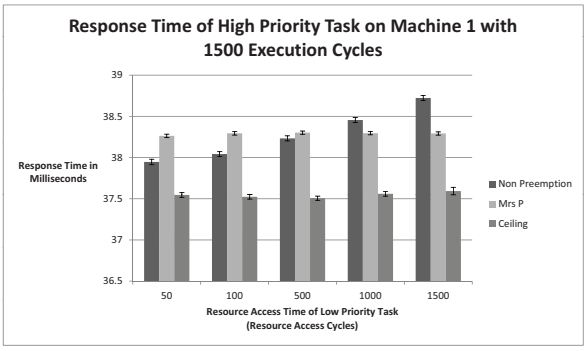

Fig. 3. Response Time of the High Priority Task (2)

priority task on core 1, low priority task on core 2 and low priority task on core 3. We again consider the response time of the low priority task on core 3. As can be seen, in Figure 4, using the FIFO ceiling algorithm the response time reflects the execution of the two high priority tasks on cores 1 and 2. With MrsP, the response time remains constant. However, now there is a more noticeable difference between MrsP and Non-preemption due to the extra migrations.

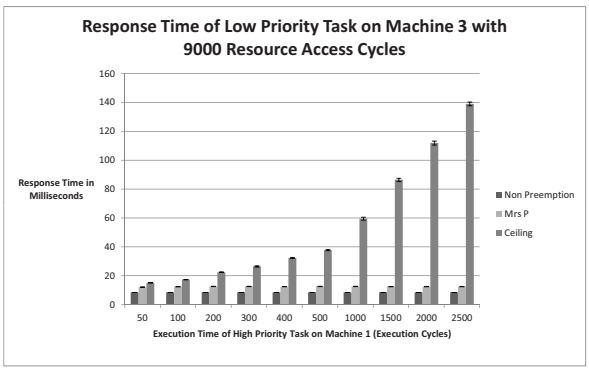

Fig. 4. Response Time of the Low Priority Task Following 2 Migrations

IX. Conclusions

In this paper we have motivated a resource control protocol for multiprocessor systems that leads to effective schedulability analysis that has an identical form to response-time analysis for single processor systems. For each resource, the number of possible parallel requests is computed. This parameter is used to bound the impact of serialising access to the resource over these parallel requests. This in turn leads to an extension of the access time for these resources, but this must be balanced by the benefits obtained from parallel execution of the application's tasks.

MrsP is a 'busy waiting' algorithm. A request for a resource results in the priority of the requesting task being raised to a local ceiling. Requests are then processed in a FIFO order. The maximum length of the FIFO queue is bounded by the number of processors on which requesting tasks can execute. The key property of MrsP is that any task, while waiting to gain access to a resource, can execute the requests of other tasks (on other processors) that are ahead of it in the FIFO queue. This property is essential as it prevents the degradation of performance that would otherwise result if the task with the resource lock were locally preempted.

Whilst motivated by a desire to define a protocol that has the expected (and minimal) impact on schedulability, it is important that the proposed protocol is implementable and has, at least, the potential to have acceptable overheads in practice. We have defined two possible means of realising MrsP; one based on migration, the other on duplication (of the critical section associated with the resource). For the migration approach we have developed a prototype implementation. This has demonstrated that the protocol is indeed implementable, and that it performs as expected when compared with some previously published schemes. In particular, it provides for a bounded impact on both high and low priority tasks. For low priority tasks sharing a resource there can be no better protocol than non-preemption, but the impact on high priority tasks can be significant. However, for protocols that involve spinning at some ceiling level the impact on low priority tasks can be unbounded as preemption from higher priority tasks will prevent progress for low priority tasks executing on cores where preemption has not occurred. MrsP proved a sensible compromise between these two approaches. It does not impact on higher priority tasks, and it bounds the impact on lower priority tasks.

For a given application on a given platform there can be no simpler resource control protocol that non-preemptive FIFO spinning and execution. All RTOS induce some non-preemptive blocking – the $\hat{b}$ term from equation (2). If tasks can be assigned to processors so that non-preemptive usage for all resource (application and RTOS) leads to schedulability then that is ideal. However, for many applications short deadline tasks will not be able to cope with a long sequence of application-induced non-preemptive sections that come from low priority tasks sharing resources. A non non-preemptive protocol is required, but one that does not excessively punish low priority tasks. This is as true for multiprocessor systems as it is for single processor systems where the use of a priority ceiling protocol (rather than non-preemption) is standard. The main contribution of this paper is to provide such a protocol.

Acknowledgements Richard Fuller for his help with the Linux implementation. Jim Anderson and the group at UNC for useful comments on MrsP and multiprocessor protocols in general. Charlie Lin for his input into Section II of this paper.

References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] T. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.

[3] T. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[4] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56. IEEE Computer Society, 2007.

[5] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS)*, pages 49–60, 2010.

[6] B. Brandenburg and J. Anderson. Real-time resource sharing under cluster scheduling. In *Proc. EMSOFT*. ACM Press, 2011.

[7] B. Brandenburg and A. Bastoni. The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors. In *Proc. of the 14th Real-Time Linux Workshop (RTLWS 2012), Real-Time Linux Foundation*, pages 67–86, 2012.

[8] R. Davis and A. Burns. Resource sharing in hierarchical fixed priority preemptive systems. In *Proceeding IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[9] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1 –35:44, 2011.

[10] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 90–99, 2010.

[11] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd RTSS*, pages 73–83, 2001.

[12] M. Hohmuth and M. Peter. Helping in a multiprocessor environment. In *Proc. of Second Workshop on Common Microkernel System Platforms*, pages 223–232, 2001.

[13] S. Lin, A. Burns, and A. Wellings. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. *Concurrency and Computation: Practice and Experience*, 2012.

[14] J. Lozi, G. Thomas, G. Muller, and J. Lawall. The remote core lock (RCL): Can migrating the execution of critical sections to remote cores improve performance? In *EuroSys11*, 2011.

[15] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[16] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[17] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. of the 10th International Conference on Distributed Computing*, pages 116–125, 1990.

[18] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[19] H. Takada and K. Sakamura. A novel approach to multiprogramming multiprocessor synchronization for real-time kernels. In *Proc. 18th IEEE Real-Time Systems Symposium*, pages 134–143, 1997.

[20] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proc. of ECRTS*, pages 223–232, 2012.