Scheduling Sporadic Tasks on Multiprocessors with Mutual Exclusion Constraints

Arvind Easwaran
CISTER/IPP-HURRAY,
Polytechnic Institute of Porto, Portugal
aen@isep.ipp.pt

Björn Andersson
CISTER/IPP-HURRAY,
Polytechnic Institute of Porto, Portugal
bandersson@dei.isep.ipp.pt

Abstract

Consider the problem of scheduling sporadic tasks on a multiprocessor platform under mutual exclusion constraints. We present an approach which appears promising for allowing large amounts of parallel task executions and still ensures low amounts of blocking.

1 Introduction

Typically, a real-time system is comprised of several tasks that execute on a shared computing platform comprised of (i) one or more *processors* and (ii) other shared *resources* (such as shared data-structures, shared I/O devices), where the latter must be managed under *mutual exclusion*, that is, at every instant, at most one task may hold the resource.

Managing resources that must be accessed under mutual exclusion has always been an important topic in the design of real-time systems because mutual exclusion can cause an extra delay to a task that requests a resource that is already in use by another task. Today, however, with the increasing use of multicores, this problem is accentuated. The reason is that the trend in development of multicore processors is to increase the core-count but maintain the processor speed. A task which only needs a processor to execute can typically do so without much delays because the total amount of processing capacity is so large. But a task which needs a processor and some other resources may experience a large delay because the time for the task that held the resource to finish its operation may be large (since the trend is to not increase the speed of processors).

Ensuring mutual exclusion is typically straightforward. It is common that the system designer creates a binary *semaphore* [7] for each resource and initializes this semaphore with the value one. A task requesting a resource must first perform *wait* on the semaphore corresponding to the requested resource. The task can then use the resource and when the task desires to no longer use it, the task performs *signal* on the semaphore corresponding to the resource.

Mechanisms for achieving mutual exclusion interacts with a real-time scheduler however and dealing with those interactions and analyzing those interactions are non-trivial. It is well known that a negative phenomena, called *priority-inversion*, can occur to priority-based real-time scheduling algorithms if the priority of a task is not adjusted at run-time to take into account the interactions between tasks that share resources. Fortunately, the real-time systems computing community has created a family of successful protocols for priority-based scheduling in the context of mutual exclusion constraints [14, 1] where tasks are scheduled on a single processor and these protocols have been adapted for multiprocessors as well [13, 12, 5, 10, 8, 6, 9, 4]. Unfortunately, these protocols adapted to multiprocessors tend to be very conservative when deciding whether to grant a resource to a task. This can unnecessarily cause a task to be prevented from executing and hence the poor performance of such protocols present a major roadblock for efficiently exploiting the parallel processing capacity of multicores in future real-time applications. Therefore, in this paper, we discuss ideas on the design of a new protocol for resource sharing that (i) ensures mutual exclusion, (ii) limits priority-inversion, and (iii) allows a large degree of parallel execution.

The remainder of this paper is organized as follows. Section 2 discusses our system model and related work. Section 3 gives an understanding of the problem on both uniprocessors and multiprocessors. Section 4 shows a new protocol which offers more parallelism. Finally, Section 5 gives conclusions and open questions.

2 System model and background

2.1 System model

Task model. For the purposes of this paper, we assume that jobs are generated by sporadic tasks [11], and are scheduled on a multiprocessor platform comprised of m identical processors. A real-time system with shared resources is specified using p non-preemptable shared resources $R_1, \ldots, R_p$, and n sporadic tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$. Each sporadic task τ_i $(1 \leq i \leq n)$ is characterized as (T_i, C_i, D_i), where T_i denotes the minimum inter-arrival time, C_i the worst-case execution time, and D_i the relative deadline. Each job of task τ_i requires C_i units of processing capacity within D_i time units from its release, and this processing capacity must be supplied sequen-

tially, *i.e.*, the job cannot be scheduled on more than one processor at any given time instant. Further, any two successive jobs of this task must be released at least T_i time units apart.

Shared resources. Jobs can issue requests for exclusive access to the shared resources $R_1, \ldots, R_p$. A request for resource R_j by a job J of task τ_i is said to be *granted* as soon as J *holds* the resource. Associated with such requests is the worst-case duration of time for which jobs of τ_i require resource R_j, denoted as $CS_{i,j}$. Note that $CS_{i,j}$ is the largest critical section among all requests of jobs of task τ_i for resource R_j. Once job J has executed for the amount of time it requires R_j, the request is said to be *complete* and the resource is said to be *released*. If a request for a resource cannot be immediately granted, then J is said to be *blocked* on R_j. And the job that holds resource R_j is said to be *directly blocking* J. Request for a resource R_k is said to be *nested* within request for another resource R_j iff R_k's request is issued after R_j's request but before R_j is released. In this paper we assume that all resource requests are *perfectly-nested*. If some job first requests resource R_j and then resource R_k while holding R_j, then it will first release R_k followed by R_j in this nesting. Further, an *outermost* resource request R_j is a request which is not preceded by any another resource request whose matching release is after the request for R_j, *i.e.*, R_j is the first resource requested in some nested resource access.

Multiprocessor scheduling. In general, studies on real-time multiprocessor scheduling theory can fall into two categories: *partitioned* and *global* scheduling. Under partitioned scheduling, each task is statically assigned to a single processor, and uniprocessor scheduling algorithms are used to schedule tasks. A critical section can then be termed as *local* if all tasks ever accessing the shared resources in this critical section are assigned to the same processor, or *global* otherwise. In this paper we focus on global scheduling algorithms. Under global scheduling, tasks are allowed to migrate across processors, and algorithms that simultaneously schedule on all the m processors are used. In this paper, for simplicity of presentation, we focus on fixed-priority global scheduling and assume that every job of τ_i has higher priority than every job of τ_j for all $i < j$. The protocol described in Section 4 can be easily extended for dynamic-priority global schedulers with static task preemption levels (see preemption level description in [1]).

2.2 Related work

In work on uniprocessor resource sharing, the *priority ceiling protocol* (PCP) [14] and the stack-based *resource allocation protocol* (SRP) [1] have received much attention. For multiprocessor systems, there has been a growing interest in the area of resource synchronization. Rajkumar *et. al.* were the first to propose a semaphore-based protocol for resource sharing on multiprocessors [13, 12]. Two variants of PCP were presented by them for systems that use partitioned, fixed-priority scheduling. Several protocols related to multiprocessor PCP have since been proposed for systems scheduled under partitioned, dynamic-priority (EDF) scheduling. Chen and Tripati [5] proposed two extensions to the basic protocol, but these extensions were only valid for periodic[1] (and not sporadic) task systems. Further, global critical sections were assumed to be non-preemptable and nesting was not allowed between global and local critical sections (each can be separately nested however). In later work, Lòpez *et. al.* [10] presented an implementation of SRP for partitioned EDF. However, this study required that tasks sharing resources be assigned to the same processor. Recently, Gai *et. al.* [8] also presented an implementation of SRP for partitioned EDF and compared it to PCP. They have implemented a FIFO-queue based spin-lock for global critical sections, which has the potential to waste processing time (tasks can busy-wait for other tasks accessing global critical sections). Further, accesses to different global critical sections are not allowed to be nested, and these critical sections are executed in a non-preemptive manner. The latter requires modifications in the kernel to disable preemptions.

In resource synchronization under global scheduling algorithms, there have been a few recent studies [6, 9, 4]. Under global EDF, Devi *et. al.* [6] proposed a FIFO-queue based spin-lock implementation for non-nested critical sections. They also modified the global EDF scheduler to enforce non-preemptive critical sections. Holman and Anderson [9] have proposed various techniques for implementing non-nested critical sections under Pfair [2] global scheduling. They allow FIFO-queue based access to locked resources and present different techniques for handling short and long critical sections. Flexible Multiprocessor Locking Protocol (FMLP), proposed by Block *et. al.* [4], can be used under partitioned EDF, global EDF, and Pfair scheduling. They handle short critical sections using FIFO-queue based spin-locks, and long critical sections using priority inheritance similar to PCP. Under partitioned scheduling, global critical sections are required to be non-preemptive. Further, nested critical sections are required to have group locks (separately for short and long sections), thus negating the benefits of nesting.

3 Understanding the problem

We will first (in Section 3.1) see the well-known problem of priority-inversion and the idea of how this is solved in the context of uniprocessor scheduling. We will then see (in Section 3.2) that transferring this idea to multiprocessor scheduling can cause a limitation in efficient use of available processing capacity (platform parallelism).

3.1 Uniprocessor systems

Suppose task set $\mathcal{T}$ is scheduled on a single processor using fixed-priority scheduling. It is assumed that the priority of a job is not affected by whether the job is holding a resource or not. Figure 1 shows an example of three jobs where J_1 and J_3 request some shared resource R and job J_2 never requests this

[1] A periodic task is similar to a sporadic task, except that T_i now denotes the exact inter-arrival time instead of minimum.

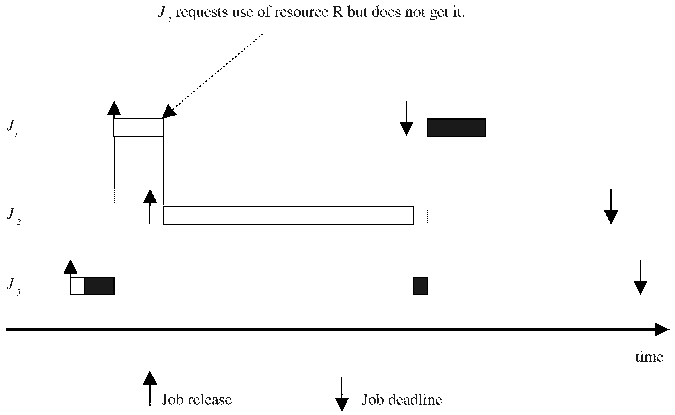

Figure 1. Priority inversion

resource. It is assumed that J_1 has higher priority than J_2 and J_2 has higher priority than J_3. First J_3 is released. It executes and then it requests resource R. J_3 is granted the resource R and it continues executing holding the resource. Before it releases R however, J_1 is released and since J_1 has higher priority, it preempts the execution of J_3. Job J_1 continues to execute and then requests resource R. This request is denied since R is held by job J_3, *i.e.*, job J_1 is blocked and cannot execute further. In the meantime, since job J_2 was released for execution and it has higher priority than J_3, it is scheduled by the dispatcher. This job executes for a long time and during its execution the deadline of job J_1 expires.

In the above example, even when a higher priority job J_1 is blocked on a shared resource R held by a lower priority job J_3, a medium priority job J_2 is allowed to execute and eventually delay the execution of job J_1. Although it is inevitable that J_1 must block until J_3 releases resource R, J_1 must not be required to wait for job J_2 to finish executing because J_2 is not holding any resource required by J_1.

The research community has invented protocols to reduce this effect (priority-inversion), and those protocols give jobs that hold shared resources temporarily a higher priority. Figure 2 shows the same jobs as in Figure 1 but now the priority of job J_3 is promoted when it holds resource R. In this way, we can see that job J_1 will meet its deadline, because job J_2 is not allowed to execute inbetween. There are different ways to promote the priority of a job that holds a resource; (i) the job could be scheduled non-preemptively or (ii) the job could inherit (transitively) the maximum priority among all jobs that are presently blocked on the same resource or (iii) the job could be assigned the ceiling priority of the resource[2]. The latter approach can be combined with a test that is performed whenever any job requests a shared resource; the job is granted access to the resource iff it has higher priority than the ceiling of all resources that are presently locked. This protocol is known as the Priority-Ceiling-Protocol (PCP) [14]. This protocol avoids deadlocks, and ensures that a job is only ever blocked once during its entire execution, even if it requests many resources in a nested manner.

Figure 3 shows an example of the operation of PCP. There are three jobs J_1, J_2 and J_3, where J_1 has higher priority than J_2 and J_2 has higher priority than J_3. Job J_3 uses one resource R_1; job J_2 uses another resource R_2; and job J_1 uses both resources R_1 and R_2. When job J_2 arrives it will not preempt the execution of J_3, because J_3 holds a resource (R_1) with priority ceiling (equal to the priority of J_1) higher than the priority of J_2. This priority promotion of job J_3 is necessary, because otherwise job J_1 would be blocked by both J_2 and J_3.

3.2 Multiprocessor systems

Suppose we have the same scenario as in Figure 1, but now the jobs are scheduled on a multiprocessor platform comprised of 2 processors. Then, it is possible to schedule job J_2 without having to preempt the execution of job J_3, and therefore J_2 will not interfere with the execution of job J_1. This brings us to the question, "Is it okay to schedule a medium priority job as long as it does not preempt any resource holding lower priority job?". Although the answer seems positive from the previous example, this is not true in all cases. It is actually not just the preemption that causes a deadline miss; the preempting job may request and be granted a resource and then this resource must be released before some high priority job can use it (example in Figure 3). Therefore we will now pose the question, "Should a request for a resource be granted?", in a multiprocessor scheduling context.

Let us consider three jobs J_1, J_2 and J_3 as in Figure 3, but now scheduled using a global fixed-priority strategy on a multiprocessor platform comprised of 3 identical processors. Figure 4 shows a scenario where J_1 misses its deadline because J_2 was granted a resource (R_2) and J_1 requested resource R_2 before J_2 had released it. Clearly, the resource sharing protocol in this scenario takes the wrong decision of granting resource R_2 to job J_2. Note that scheduling job J_2 is by itself not a wrong decision; however granting it resource R_2 at a time

[2]The ceiling priority of a resource R is the maximum priority among all jobs that might ever request the resource R. See [14] for more details on ceiling priorities.

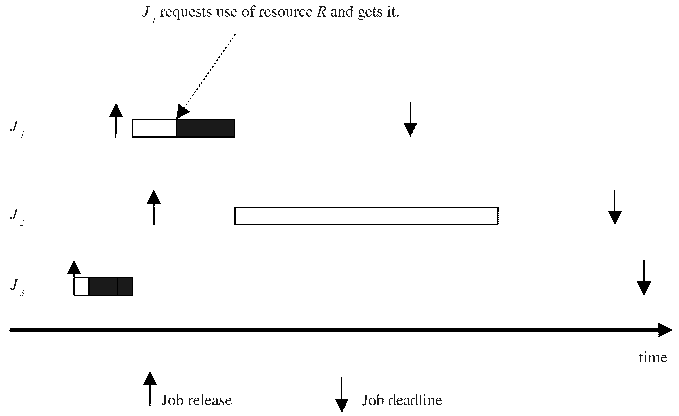

Figure 2. Priority inversion is reduced by temporarily giving J_3 higher priority

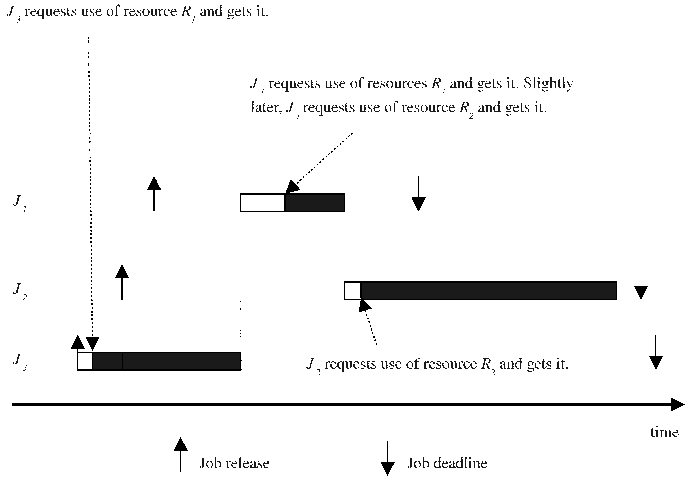

Figure 3. Behavior of Priority Ceiling Protocol (PCP)

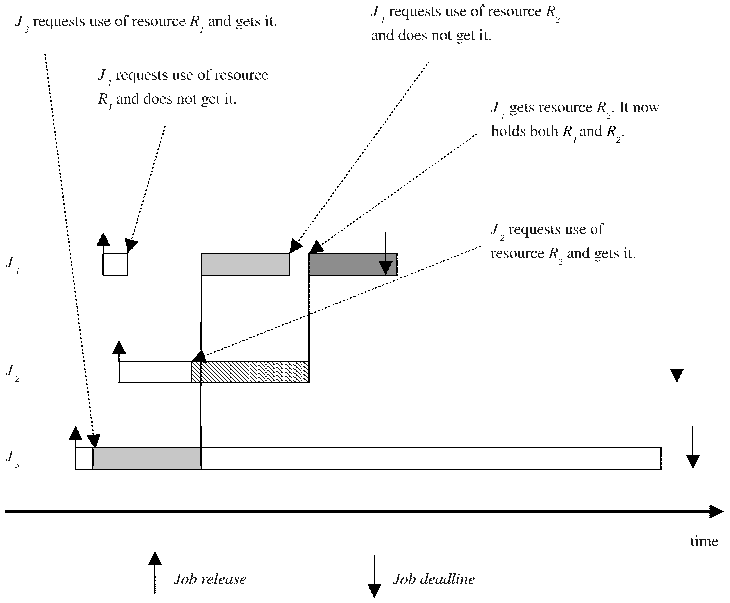

Figure 4. Global multiprocessor scheduling (Failure)

when it was known that job J_1 will also request the same resource is a wrong choice in this scenario[3]. Essentially, job J_1 suffers blocking twice within the same nested resource access; once while waiting for resource R_1 and again while waiting for resource R_2. An improved resource sharing scenario in which job J_2 is denied access to resource R_2 is illustrated in Figure 5.

This then begs the question, "When should a request for shared resource be granted?". A very safe approach would be to grant access to only one resource at a time, but this would limit parallelism. And this limited parallelism would imply that more work must be done at later times, which in turn can cause deadline misses. Another approach would be to use a PCP-like protocol (as in uniprocessors) and decide that job J_2 should be denied resource R_2, because J_2 does not have higher priority than the ceilings of all locked resources (namely R_1). But this can also unnecessarily limit parallelism resulting in the aforementioned performace drawback. In Figure 4, if job J_2 would have released resource R_2 just prior to when job J_1 requested access to the same resource, then it would not have affected the finishing time of job J_1. Thus we can see that a resource request from a medium priority job can be granted if the resource is released before any other higher priority job requests it. Or more generally, ***a resource request can be granted as long as the maximum blocking time suffered by any higher priority job is guaranteed to be within pre-defined bounds***. In fact, in the next section, we will present a resource sharing protocol based on this idea. It allows parallelism (=granting requests) as much as possible, yet keeping the blocking time within limits.

4 BHP resource sharing protocol

The main idea. From the discussion in the previous section we can draw the following conclusions about the design of a protocol which avoids priority inversion and allows a large degree of parallel execution:

- A priority-inheritance mechanism should be used in order to avoid priority inversion but a PCP-like mechanism should not be used (because it would restrict parallel execution too much).
- If a high-priority task requests to execute but it does not request a shared resource then this task should be allowed to continue to execute.
- There should be a mechanism for preventing deadlock. (This is needed since we do not use PCP.)
- For each task-resource pair, there should be an associated counter variable. This counter specifies the amount of blocking that the task can tolerate to be blocked when requesting the resource. For every resource request, the protocol should check so that granting the request does not violate any tolerated blocking of any other task-resource pairs.
- If a task is blocked for one time unit because the task requested a resource then the corresponding counter of this task-resource pair should be decremented by one (since the amount of tolerable blocking is one time unit less).

In this section, we will create a protocol based on these ideas and this protocol will avoid priority inversion and allow a large degree of parallel execution. We will first present notations that is needed. Then present the dispatching algorithm for global scheduling; it uses the counters as mentioned above. We will then show how the counters as updated and discuss subtle issues with the protocol.

Notations. We use the following notations.

- t: Denotes the current time instant.
- LPB_i: Denotes the *L*ower *P*riority *B*locking for jobs of task τ_i. Our protocol guarantees that for each nested resource access (the entire nesting) by jobs of task τ_i, the maximum time for which this job will be blocked by lower priority jobs is at most LPB_i. If a job has two completely separate nested resource accesses during its execution, then it will incur a maximum blocking of $2LPB_i$. Our protocol guarantees a value of $\max\{CS_{k,l}\}$ for LPB_i, where $k > i$ and l ranges over all resources that jobs of τ_i access.
- $MTR_{k,l}$: *M*inimum *T*ime to *R*equest resource R_l counting from the start of the nesting, among all non-outermost accesses to this resource by jobs of task τ_k. For example, suppose jobs of τ_k request resource R_l in three nestings during their execution; 1) R_j requested and then R_l with a minimum gap of 10 time units, 2) R_l alone requested, and 3) R_j requested and then R_l with a minimum gap of 5 time units. Then, $MTR_{k,l}$ in this case is 5.
- $\lfloor R_l \rfloor_k$: For each task τ_k and each resource R_l, $\lfloor R_l \rfloor_k$ denotes the maximum blocking (in future) that the currently active job of τ_k can incur in its current resource nesting. If the job is currently not in any nesting or if $\lfloor R_l \rfloor_k$ is currently irrelevant, then it is set to ∞. The value is initialized to ∞, and only updated by our protocol.
- PTY_i: Priority of jobs of task τ_i at the current time instant. PTY_i is initialized to i, but can be temporarily modified by our protocol (PIP-like updates). A job of task τ_i has higher priority than a job of task τ_j iff $PTY_i < PTY_j$, or $PTY_i = PTY_j$ and $i > j$[4].

BHP Protocol. The *B*ounded Blocking with *H*igh *P*arallelism resource sharing protocol is given by Algorithm 1, and the update to $\lfloor R_l \rfloor_k$ is performed by Algorithm 2. Both these algorithms are executed at each time instant t, with Algorithm 2 being executed first.

[3] J_1 was released before J_2 was granted access to R_2, and at that time we knew that J_1 needs resource R_2.

[4] The last condition ensures that any lower priority job of task τ_i that is promoted to level j has higher priority than jobs of task τ_j

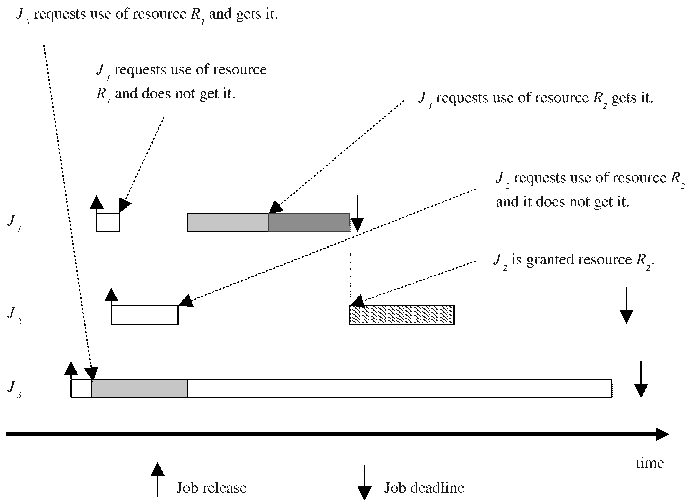

Figure 5. Global multiprocessor scheduling (Success)

Algorithm 1 Global scheduling with resource sharing

```
1:  n_assigned ← 0
2:  for each ready job J in priority order (based on PTY_i) do
3:      if n_assigned < m then
4:          if J is not requesting any resource then
5:              Execute job J.
6:              n_assigned ← n_assigned + 1
7:          else
8:              Let R_j denote the resource requested.
9:              if all resources in the nesting to which
10:                 this request belongs are unlocked then
11:                 if ∀k : ⌊R_j⌋_k ≥ CS_ij or
12:                   PTY_i < PTY_k then
13:                     Execute J and set PTY_i equal to the
14:                     smallest PTY_k such that ⌊R_j⌋_k
15:                     has a finite value (this update to PTY_i
16:                     is reset when resource R_j is released).
17:                     n_assigned ← n_assigned + 1
18:                 end if
19:             end if
20:         end if
21:     end if
22: end for
```

We now explain the BHP protocol using examples illustrated in Figure 6. The figure illustrates two scenarios separated by the thick vertical line. In both the scenarios job J_1 has higher priority than job J_2 and J_2 has higher priority than job J_3, and these jobs are scheduled on 2 identical processors. Further, job J_1 requests resource R_1, job J_2 requests resource R_1 followed by resource R_2 in a nested manner, and job J_3 requests resource R_2. In the scenario on the left, job J_2 locks resource R_1 and then job J_1 blocks because it needs R_1. Then, job J_3 arrives and requests to lock resource R_2. This request is granted because the maximum time for which J_3 will lock R_2 is such that it will not delay the execution of job J_2 (R_2 will be released before t_1 in the figure). In fact, the protocol will allow J_3 to lock R_2 even if it delays the execution of J_2, as long as this delay does not increase the blocking time of J_1 beyond LPB_1. This scenario is depicted on the right-hand-side of the figure (interval $(t_2, t_3]$ is bigger than interval $(t_3, t_4]$). In summary, when J_3 requests resource R_2, it is granted access as long as the total blocking suffered by J_2 is such that it does not unnecessarily increasing the blocking of J_1. The adjustment of blocking parameters in Lines 24 and 28 of Algorithm 2 ensures this property. The BHP protocol thus allows lower priority jobs to lock resources even when dependent (through nesting) resources are locked by higher priority jobs, and thereby improves parallelism when compared to other existing approaches.

It can be shown that the BHP protocol prevents deadlocks (due to check in Line 9 of Algorithm 1), and ensures that the maximum lower priority blocking suffered by any job of task τ_i is LPB_i. The latter is true because of the following reasons.

- The protocol allows a lower priority job to lock a resource iff it does not violate the blocking constraint in Line 11 of Algorithm 1. This constraint checks, for each higher priority task τ_k, whether the maximum blocking time $CS_{i,j}$ is smaller than permitted blocking time of resource R_j ($\lfloor R_k \rfloor_j$).

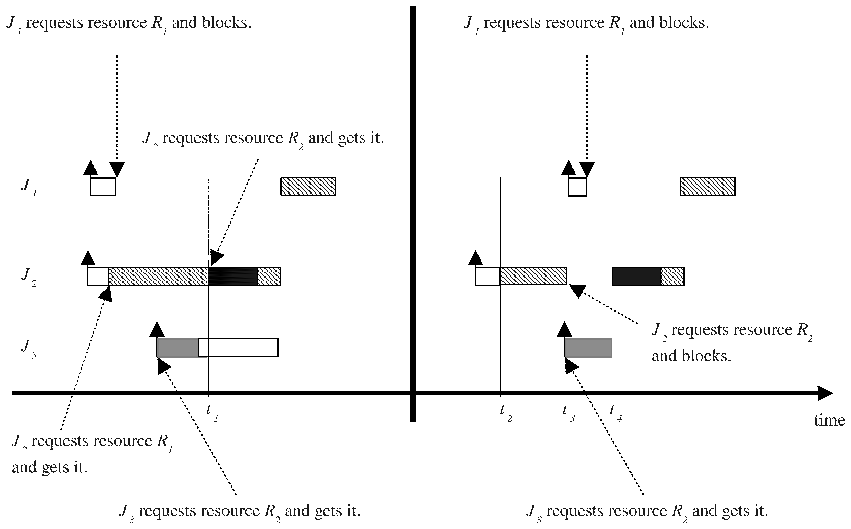

Figure 6. Example illustrating the BHP protocol

Algorithm 2 Update rules for $\lfloor R_l \rfloor_k$

1: **if** A job of task τ_i performs an *outermost* request for
2: resource R_j (first request of a nested access) **then**
3: $\lfloor R_j \rfloor_i \leftarrow LPB_i$.
4: **if** R_j is currently locked by a job of task τ_k and
5: $PTY_k > PTY_i$ **then**
6: $PTY_k \leftarrow PTY_i$ (this update to PTY_k is reset
7: when resource R_j is released).
8: **end if**
9: $\lfloor R_l \rfloor_i \leftarrow MTR_{i,l}$ for each non-outermost resource R_l
10: in this nested access.
11: **for** each non-outermost resource R_l in this
12: nested access **do**
13: **if** R_l is currently locked by a job of task τ_k and
14: $PTY_k > PTY_i$ **then**
15: $PTY_k \leftarrow PTY_i$ (this update to PTY_k is reset
16: when resource R_l is released).
17: **end if**
18: **end for**
19: **end if**
20: **if** a job of task τ_i is granted access to a resource R_j (in response to an earlier request) **then**
21: $\lfloor R_j \rfloor_i \leftarrow \infty$
22: **end if**
23: **if** A job of task τ_i is blocked in the interval $(t-1, t]$ **then**
24: $\lfloor R_l \rfloor_i \leftarrow \lfloor R_l \rfloor_i - 1$, for all l s.t. $\lfloor R_l \rfloor_i \neq \infty$.
25: **else**
26: **if** A job of task τ_i is directly blocking some job in the
27: interval $(t-1, t]$ **then**
28: $\lfloor R_l \rfloor_i \leftarrow \lfloor R_l \rfloor_i - 1$, for all l s.t. $\lfloor R_l \rfloor_i \neq \infty$.
29: **end if**
30: **end if**

- The initialization of blocking parameters (Lines 3 and 9 in Algorithm 2) ensure that the total direct blocking that a lower priority job can induce on a job of task τ_i is bounded by LPB_i. Further, the updates to these parameters, in Lines 24 and 28 of Algorithm 2, ensure that the total blocking (direct and indirect) incurred by jobs of τ_i is bounded by LPB_i.

Although the BHP protocol prevents deadlocks, ensures bounded blocking, and improves the parallelism in task executions when compared to existing studies, it incurs overheads in terms of memory requirements for $\lfloor R_l \rfloor_k$ ($\mathcal{O}(np)$) and online time-complexity for updates to $\lfloor R_l \rfloor_k$ ($\mathcal{O}(np)$).

5 Conclusions and open questions

In this paper, we have discussed that there is a tradeoff between blocking and parallelism, and we have proposed the BHP protocol which allows as much parallelism as possible, keeping blocking within pre-defined limits. We may note that we are not the first ones to propose that a request for a resource should undergo a check, to calculate the time when the resource will be released. In fact, SIRAP [3] (a protocol for hierarchical scheduling) used such a test to decide if a job which requests a resource will finish execution before its budget expires, and if the answer is no then the request is denied.

Although the BHP protocol addressed some issues concerning the efficient use of parallelism in task executions, some open questions still remain. One of them is "Moving from uniprocessors to multiprocessors, whether it is still relevant to treat processors in a special manner when compared to other shared resources?". In multiprocessors, there is a very clear trade-off between mutually exclusive access to shared resources and ability to exploit processing parallelism. Then, it would be interesting to consider processors as just another

shared resource (although preemptable), and integrate their scheduling directly into the resource sharing protocol. Another question is "How to integrate the loss of parallelism due to shared resources in schedulability analysis?". There are two factors leading to loss of parallelism; blocking from lower priority jobs and blocking from higher priority jobs[5]. The former is accounted for in the blocking factor (LPB in our case). However, accounting for the latter is still an open problem.

References

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

[2] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. pages 279–288, 2007.

[4] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of Real-time and Embedded Computing Systems and Applications Conference*, pages 47–56, 2007.

[5] C.-M. Chen and S. K. Tripathi. Multiprocessor priority ceiling based protocols. Technical report, 1994.

[6] Um. C. Devi, H. Leontyev, and J. H. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *Proc. of Euromicro Conference on Real-Time Systems*, pages 75–84, 2006.

[7] E. W. Dijkstra. Co-operating sequential processes. *Programming Languages*, pages 43–112, 1968.

[8] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multiprocessor systems-on-a-chip. In *Proc. of IEEE Real-Time Systems Symposium*, page 73, 2001.

[9] P. Holman and J. H. Anderson. Locking in pfair-scheduled multiprocessor systems. In *Proc. of IEEE Real-Time Systems Symposium*, page 149, 2002.

[10] J. M. López, J. L. Díaz, and F. D. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Journal of Real-Time Systems*, 28(1):39–68, 2004.

[11] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, PhD Thesis, Department of Computer Science, Massachusetts Institute of Technology (MIT), 1983.

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

[13] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. pages 259–269, 1988.

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

[5] Higher priority blocking arises when processors are idle because higher priority jobs have locked resources required by lower priority jobs.