Towards Resource Sharing under Multiprocessor Semi-Partitioned Scheduling

Sara Afshar, Farhang Nemati, Thomas Nolte
Mälardalen University, Västerås, Sweden
Corresponding author: farhang.nemati@mdh.se

***Abstract*—Semi-partitioned scheduling has been subject of recent interest, compared with conventional global and partitioned scheduling algorithms for multiprocessors, due to better utilization results. In semi-partitioned scheduling most tasks are assigned to fixed processors while a low number of tasks are split up and allocated to different processors. Various techniques have recently been proposed to assign tasks in a semi-partitioned environment. However, an appropriate synchronization mechanism for resource sharing between tasks in semi-partitioned scheduling has not yet been investigated. In this paper we propose two methods for handling resource sharing under semi-partitioned scheduling in multiprocessor platforms. The main challenge is to handle the resource requests of tasks that are split over multiple processors.**

I. Introduction

Research studies on real-time scheduling techniques suitable for multiprocessor systems has largely increased due to the dramatic rise of interest towards usage of multi-core technology in embedded systems. The shift towards multi-core technology has emerged the need for real-time scheduling algorithms and resource sharing protocols, which support real-time applications on multiprocessors. Two major conventional algorithms developed for scheduling real-time tasks on multiprocessors are categorized as global and partitioned scheduling. In partitioned scheduling each task is statically assigned to a single processor on which all of its jobs will execute. In global scheduling a global ready queue is used to store all ready tasks in the system.

Semi-partitioned scheduling is a mix between the pure partitioned and global scheduling approaches, which has been first introduced by Anderson et al. in [1]. Semi-partitioned scheduling extends partitioned scheduling by allowing a low number of tasks to be split among different processors and thereby improving the schedulability, while other tasks in the system are allocated to fixed processors. Similar to partitioned scheduling, semi-partitioned scheduling utilizes separate ready queues for each processor in which the individual scheduler of each processor manages ready tasks from the ready queue to access the processor capacity. Different task allocation methods have been proposed in [2], [3], [4], [5]. Guan et al. in [5] allow the utilization of task sets to be as high as the utilization bound of Liu and Layland's *Rate Monotonic Scheduling* (RMS) for any task set.

A semi-partitioned scheduling approach consists of three parts: 1) the partitioning algorithm which determines how to allocate tasks, or split them if required, among processors, 2) the scheduling algorithm which specifies how to schedule the tasks assigned to each processor, and 3) the synchronization protocol to manage sharing of mutually exclusive resources between tasks which is the focus of this paper. Execution of tasks on multiprocessors can cause longer blocking delays comparing to executing the tasks on uniprocessors due to delays caused by other processors in the system. Therefore, the need of an efficient real-time synchronization protocol for accessing the shared resources is magnified under semi-partitioned scheduling. For this purpose we present two solutions in this paper for handling shared resources undr semi-partitioned scheduling.

The goal of a synchronization protocol is to bound the waiting times of tasks which share resources in the system. Vast amount of work has been done on resource sharing under partitioned scheduling algorithms. Rajkumar et al. proposed the Multiprocessor Priority Ceiling Protocol (MPCP) in [6]. The Multiprocessor Stack Resource Policy (MSRP) introduced by Gai et al. in [7] is another resource sharing technique for multiprocessors. The Flexible Multiprocessor Locking Protocol (FMLP) proposed by Block et al. in [8] for partitioned and global scheduling algorithms and later the partitioned FMLP was extended by Brandenburg and Anderson in [9] for fixed priority scheduling. Another locking protocol for handling resource sharing in multiprocessors is $O(m)$ Locking Protocol (OMLP) introduced by Brandenburg and Anderson in [10]. Multiprocessor Synchronization protocol for Open Systems (MSOS) is another synchronization mechanism for resource sharing among independently-developed real-time applications presented by Nemati et al. in [11].

Inspired by the previous protocols we have investigated two methods for handling resource requests under semi-partitioned scheduling regardless of the partitioning algorithms. In the rest of this paper we present the proposed methods and analyze the relevant blocking durations.

II. System Model

In this section we introduce the system and task model. The multiprocessor platform consists of a task set comprising of n periodic tasks $\{\tau_1, \tau_1, ..., \tau_n\}$ which is running on m processors $\{P_1, P_1, ..., P_m\}$. Each task τ_i conforms to the (C_i, T_i, ρ_i) model where C_i is the worst-case execution time, T_i is the period and ρ_i is the priority of the task τ_i. Tasks have implicit deadline, i.e., T_i is also τ_i's relative deadline. Without loss of generality, task τ_i is assumed to have priority higher than that of task τ_j, if $\rho_i > \rho_j$.

Tasks which are assigned to one processor are called non-split tasks and those which are split over more than one processor are called split tasks. However each single part of a

split task allocated to different processors is called a subtask of a split task. The subtasks of each split task in the system should be synchronized with each other in the sense that each subtask finishes its execution prior to its successive subtasks. This means that a subtask of a split task can not be executed before the former subtask is finished. As shown in the example in Figure 1 task τ_i has three subtasks: τ_i^1, τ_i^2 and τ_i^3 which are the first, second and third subtasks respectively of the split task τ_i. a denotes the arrival time of the task τ_i and T_i is the deadline. τ_i^2 arrives with a constant offset which is equal to the τ_i^1's worst-case response time r_i^1. Similarly τ_i^3 becomes ready to execute with a constant offset equal to the worst-case response time of τ_i^2. Yet the deadline of subtask τ_i^3 and the whole task is T_i. Accordingly, we present the subtasks of split tasks except the first subtask with (C_i, T_i, ρ_i, O_i) where O_i determines the constant offset caused by the delay imposed from the former subtask's maximum response time. For subtasks of the split tasks, ρ_i is identical and the same as task τ_i's priority.

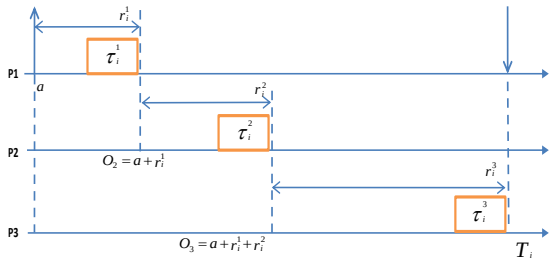

Fig. 1. Subtasks in a split task

The tasks on processor P_k share a set of resources R_{p_k} which are protected by semaphores. The set of tasks on processor P_k that request resource R_q are known as $\tau_{q,k}$. The shared resources R_{p_k} can be either the local or global resources. Local resources are only shared by tasks on the same processor, while global resources are shared by tasks on different processors. The set of local and global resources are denoted by $R_{P_k}^L$ and $R_{P_k}^G$ respectively. All resources requested by subtasks of split tasks are assumed as global resources since a critical section requesting a resource may happen in different processors. Moreover, $Cs_{i,q}$ denotes the worst-case execution time of the longest critical section in which τ_i requests resource R_q. $n_{i,q}^G$ is the number of τ_i's global critical sections (gcs) in which it requests R_q.

III. Algorithm Description

In semi-partitioned scheduling a major group of tasks is allocated to one fixed processor and a low number of tasks which can not completely fit into one processor during the allocation process will be split among different processors. Each processor has its own scheduler and ready queue in which tasks competing for the processor capacity are enqueued. The allocation mechanism, i.e., how to assign tasks to the processors in the system and how to split tasks once they can not totally fit in one processor is not investigated in this paper. Our focus in this paper is how to manage resource requests according to the fact that some tasks in the system are allocated to more than one processor. Therefore we assume that there exists an appropriate algorithm under which tasks are assigned to processors according to a semi-partitioned protocol, e.g., the approach of Guan et al. [5].

Once a processor can not dedicate enough capacity to a task, the task will be split to subtasks such that the first subtask fills the processor and the other subtasks are assigned to the next processors. In other words the processor which contains the first subtask of a split task will have no extra space for more tasks to be assigned.

Under semi-partitioned scheduling both split and non-split tasks may request mutually exclusive resources. The tasks requesting mutual exclusive resources are guarded by semaphores.

A priority-based global queue is considered for each global resource in which tasks requesting the resource are enqueued. As soon as the resource is granted to the task, it is inserted in a local priority-based queue along with all other tasks of the processor which also have been granted other global resources.

Different execution characteristics of a task such as loop iteration bounds, recursion depth bounds and infeasible paths cause different executions paths. Accordingly, the critical sections of tasks may happen at different times within the task execution time. As a general model of this behavior, this means that a critical section may occur at any time during the task execution. In the case of split tasks, the critical sections may happen in different subtasks of the split task. There is no guarantee that a specific critical section happens in the same subtask and subsequently on the same processor in different instances of the task. However, it should be noticed that a specific critical section in a split task can only occur in one of the subtasks, and once it is finished it can not happen in any other subtask of that task instance. Under this terms we suggest two algorithms for handling the resource requests in split and non-split tasks.

A. Migration-based synchronization protocol

The first algorithm is based on centralizing all critical sections happening in different subtasks on one marked processor. The marked processor is the processor that contains the subtask which can fit all critical sections of the original task. This means that every time a job in a subtask of a split task requests a resource, it will migrate to the marked processor and execute its critical section non-preemptively in that processor. In other words, the task which has requested a resource releases the source processor and migrates to the marked processor (destination), therefore other tasks in the processor can have access to the processor capacity [12]. Once the job executes its critical section it will migrate back to its original processor. Note that, requests of the subtasks on marked processor are served on the marked processor and no migration happens in this case. As a result of migrations, the subtasks will incur overhead mainly due to cache-related migration overhead [13]. This overhead is caused by additional cache misses that a job incurs when resuming execution after a migration.

As it can be seen in Figure 2 subtask τ_i^2 requesting a critical section in processor P_2 migrates to processor P_1 and after executing its critical section, it migrates back to its original processor P_2. As it can be seen, two migration overheads are produced during the execution of τ_i^2.

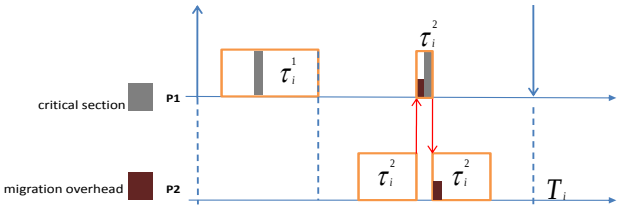

Fig. 2. Resource handling in migration-based synchronization protocol

Once each subtask of a split task is considered as an independent task, requests on global resources of all tasks are served identically. First the requesting task is enqueued in the related global prioritized queue and then when the resource is granted to the task, it is inserted in the prioritized local queue in its processor. In the case of subtasks which migrate to the marked processor, the task which has been granted the resource is inserted to the local queue of the marked processor. The tasks enqueued in the local queue are granted access to different resources and are waiting for the processor. The highest priority task in the local queue can start using its requested resource. The priority of the task which gets access to the resource is boosted to a priority higher than any priority in that processor allowing the task to execute its critical section non-preemptively. If ρ_h is the highest priority in the processor P_r the task's priority is boosted to $\rho_h + 1$ while it access a global resource. Note that, tasks are enqueued in the global and local resource queues with their original priority.

B. Non-migration-based synchronization protocol

In the second algorithm the resource requests by split tasks will be served on the same processor where the request occurs and as a result no migration happens. This implies that every time a job requests a resource, first it will be added to the global resource queue and when the resource has been granted, it will wait in its original processor local queue. Similar to the first solution each subtask of a split task is assumed to be an independent task having a constant offset caused by previous subtasks's response time. In difference with the first solution all subtasks are assumed to access global resources in their original processors and they thus incur delay to the local tasks due to the access of global resources. As it can be seen in Figure 3 subtask τ_i^2 executes its critical section on its original processor P_2. Please notice, that at any time only one subtask can be located in a global queue.

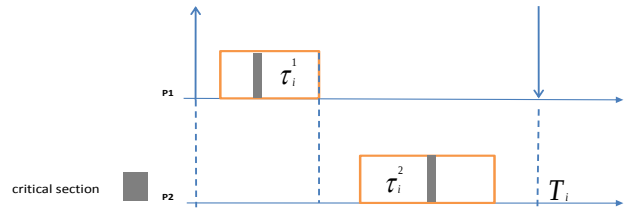

Fig. 3. Resource handling in non-migration-based synchronization protocol

Tasks in other processors may cause blocking for a specific task requesting a global resource. This blocking is called remote blocking and the processors causing this blocking are called remote processors. In case of split tasks, subtasks are treated as individual normal tasks in the schedulability analysis. In the second algorithm, more than one subtask of a split task may share resources with other tasks in the system. Therefore the number of tasks which cause remote blocking to other tasks in the system are increased in case of different subtasks requesting resources. In the first approach by centralizing critical sections of split tasks into one processor the number of tasks that cause remote blocking is decreased. However, we can not ignore the fact that overhead is increased by the first algorithm over migration of the subtasks to the marked processor.

Access to local resources in both algorithms is controlled by a uniprocessor synchronization protocol, e.g. PCP or SRP.

IV. Schedulability Analysis

In this section we present the schedulability analysis of the two proposed approaches. There are various possible situations which may cause a task to get blocked on resources by other tasks. Next, we will enumerate four possible blocking terms which a task may experience in a multiprocessor platform under semi-partitioned scheduling. We will categorize blocking terms into local and remote blocking. When the blocking is imposed in terms of local tasks, i.e., the tasks are executing on the same processor, it is called local blocking. On the other hand, the blocking caused in terms of tasks on remote processors is identified as remote blocking.

A. Local blocking due to local resources

We denote n_i^G as the number of gcs that τ_i executes before its completion. Each time a task τ_i is suspended due to a global resource it gives the chance to a lower priority task τ_j to lock a local resource which in turn may block τ_i in any of its non-gcs. This kind of blocking can happen up to n_i^G times in the case of τ_i suspending on a global resource. Additionally, according to PCP and SRP, τ_i can be blocked on a local resource by at most one critical section of a lower priority task which has arrived before τ_i. However, τ_j can release a maximum of $\left\lceil \frac{T_i}{T_j} \right\rceil$ jobs before τ_i is finished. In addition, each job can also block τ_i's current job at most up to $n_j^L(\tau_i)$ times, where $n_j^L(\tau_i)$ is the number of critical sections of task τ_j in which it requests local resources with ceiling higher than the priority of τ_i. Thus the first blocking term denoted by $B_{i,1}$ is as follows:

$$B_{i,1} = \min\{n_i^G + 1, \sum_{\rho_j < \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\} \max_{\substack{\rho_j < \rho_i \\ \wedge\, \tau_i, \tau_j \in P_k \\ \wedge\, R_l \in R_{P_k}^L \\ \wedge\, \rho_i \leq ceil(R_l)}} \{Cs_{j,l}\} \quad (1)$$

where $ceil(R_l) = \max\{\rho_i | \tau_i \in \tau_{l,k}\}$.

B. Local blocking due to global resources

Each time τ_i is suspended on a global resource a lower priority task τ_j may get access to a global resource (enters a gcs) and preempt τ_i in any of its non-gcs. τ_i may experience this kind of blocking up to $n_i^G + 1$ times due to all its resource requests besides the situation in which τ_j has arrived and entered a gcs before τ_i arrives. Similar to Section IV-A, τ_j can release a maximum of $\left\lceil \frac{T_i}{T_k} \right\rceil$ jobs before τ_i is finished and each job of τ_j can preempt τ_i's current job at most up to n_j^G

times. Hence the blocking introduced under this terms denoted by $B_{i,2}$ is calculated as follows:

$$B_{i,2} = \sum_{\substack{\forall \rho_j < \rho_i \\ \wedge\, \tau_i, \tau_j \in P_k}} \left(\min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\} \max_{R_q \in R_{P_k}^G} \{Cs_{j,q}\} \right) \tag{2}$$

C. Remote blocking

Whenever a task has to wait for global resources due to tasks on other processors, it incures a remote blocking. In our proposed algorithms tasks may experience remote blocking from two group of tasks:

1) Tasks with lower priority: It may happen that a task τ_i on a processor P_k requests a global resource R_q which has already been granted to a lower priority task τ_j on processor P_r. In this situation τ_i has to wait until τ_j releases the resource after executing its gcs on P_r. On the other hand, τ_j may wait in the local resource queue of P_r in which all tasks that have been granted a resource are waiting. As soon as τ_j is the highest priority task in the local queue, it will access R_q. In the worst-case τ_j has to wait for all higher priority tasks in P_r which are already granted access to resources other than R_q. Consequently, these tasks indirectly delay τ_i on R_q. In order to calculate the worst-case delay caused by these tasks, all delays caused by lower priority tasks on τ_i's remote processors are calculated and the maximum value is selected. This scenario may happen for each global resource request of τ_i, therefore the related blocking term which is denoted by $B_{i,3}$ is as follows:

$$B_{i,3} = \sum_{\substack{\forall R_q \in R_{P_k}^G \\ \wedge\, \tau_i \in \tau_{q,k}}} n_{i,q}^G \max_{\substack{\forall \rho_j < \rho_i \\ \wedge\, \tau_j \in \tau_{q,r} \\ \wedge\, k \neq r}} \{Cs_{j,q} + \rho_{h,j}(R'_q) \max_{\substack{\tau_t \in P_r \\ \wedge\, \rho_t > \rho_j \\ \wedge\, R_s \in R_r^G \\ \wedge\, s \neq q}} \{Cs_{t,s}\}\} \tag{3}$$

where $n_{i,q}^G$ is the number of τ_i's global critical sections in which it requests R_q and $\rho_{h,j}(R'_q)$ is the number of local tasks with priority higher than that of τ_j that share global resources other than R_q.

2) Tasks with higher priority: A task τ_i assigned to processor P_k waiting for a resource R_q in its related prioritized global queue have to wait for all higher priority tasks of other processors in the queue. On the other hand, a higher priority task τ_t assigned to processor P_r may generate several jobs up to $\left\lceil \frac{T_i}{T_t} \right\rceil$ and each job may request R_q several times during the time that τ_i waits for R_q. Each job of τ_t can block τ_i's current job up to $n_{t,q}^G$ times; where $n_{t,q}^G$ is the number of τ_t's global critical sections in which it requests R_q. Similar to the term in Section IV-C1, τ_t may also wait in the local resource queue in P_r for at most one critical section of each higher priority task which has requested a resource other than R_q. The related blocking term which is denoted as $B_{i,4}$ is calculated as follows:

$$\begin{aligned} B_{i,4} = & \sum_{\substack{\forall R_q \in R_{P_k}^G \\ \wedge\, \tau_i \in \tau_{q,k}}} \sum_{\substack{\forall \rho_t > \rho_i \\ \wedge\, \tau_t \in \tau_{q,r} \\ \wedge\, r \neq k}} n_{t,q}^G \lceil T_i/T_t \rceil (Cs_t, q \\ & + \rho_{h,t}(R'_q) \max_{\substack{\tau_j \in P_r \\ \wedge\, \rho_j > \rho_t \\ \wedge\, R_l \in R_r^G \\ \wedge\, l \neq q}} \{Cs_j, l\}) \end{aligned} \tag{4}$$

where $\rho_{h,t}(R'_q)$ is the number of local tasks with priority higher than that of τ_t, that share global resources other than R_q.

V. Migration overhead

As mentioned before in the first proposed approach the critical sections of split tasks will migrate to a specific processor (marked processor). This migration produces overhead due to migration delay. The execution time of the migrated task is inflated by per migration overhead twice. Since the migrated task incurs a delay once when it migrates to the marked processor and once it has migrated back to its original processor as it can be seen in Figure 2. Hence the migrated critical section is also inflated with one migration overhead itself. This is because, when a task is blocking due to a migrated task, besides the critical section length, it also incurs a delay caused by the migration overhead. Therefore, this overhead is included in the critical section length.

$$C_{i_{migrated}} = C_i + C_{overhead} \tag{5}$$

$$Cs_{i_{migrated},q} = Cs_{i,q} + C_{overhead} \tag{6}$$

VI. Conclusions and Future Works

In this paper, we proposed two synchronization protocols under semi-partitioned scheduling for multiprocessor platforms. The protocols handles the resource sharing between tasks among different processors in the platform where some tasks have been allocated to more than one processor as the result of the semi-partitioned allocation mechanism.

Currently, we are developing the experimental evaluations of these synchronization protocols. Furthermore, we will work on a new allocation mechanism that take the presence of resource sharing in the system into consideration, to deliver a complete solution.

References

[1] J. Anderson, V. Bud, and U. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," in *ECRTS'05*.

[2] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors," in *IPDPS'08*.

[3] K. S. and Y. N., "Semi-partitioned fixed-priority scheduling on multiprocessors," in *RTAS'09*.

[4] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *ECRTS'09*.

[5] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-priority multiprocessor scheduling with liu and layland's utilization bound," in *RTAS'10*.

[6] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *ICDCS 1990*.

[7] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform," in *RTTAS'03*.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA'07*.

[9] B. B. Brandenburg and J. H. Anderson, "An implementation of the pcp, srp, d-pcp, m-pcp, and fmlp real-time synchronization protocols in litmus rt," 2008.

[10] B. Brandenburg and J. Anderson, "Optimality results for multiprocessor real-time locking," in *RTSS'10*.

[11] F. Nemati, M. Behnam, and T. Nolte, "Independently-developed real-time systems on multi-cores with shared resources," in *ECRTS'11*.

[12] P.-C. Hsiu, D.-N. Lee, and T.-W. Kuo, "Task synchronization and allocation for many-core real-time systems," in *EMSOFT'11*.

[13] A. Bastoni, B. Brandenburg, and J. Anderson, "Is semi-partitioned scheduling practical?" in *ECRTS'11*.