Resource Sharing under Multiprocessor Semi-Partitioned Scheduling

Sara Afshar, Farhang Nemati, Thomas Nolte
Mälardalen University, Västerås, Sweden
Email: {sara.afshar, farhang.nemati, thomas.nolte}@mdh.se

***Abstract*—Semi-partitioned scheduling has become the subject of recent interest for multiprocessors due to better utilization results, compared to conventional global and partitioned scheduling algorithms. Under semi-partitioned scheduling, a major group of tasks are assigned to fixed processors while a low number of tasks are allocated to more than one processor. Various task assigning techniques have recently been proposed in a semi-partitioned environment. However, a synchronization mechanism for resource sharing among tasks in semi-partitioned scheduling has not yet been investigated. In this paper we propose and evaluate two methods for handling resource sharing under semi-partitioned scheduling in multiprocessor platforms. The main challenge addressed in this paper is to serve the resource requests of tasks that are assigned to different processors.**

I. Introduction

Research on real-time scheduling techniques for multiprocessor platforms has received a dramatic rise of attention due to large interest towards usage of multi-core technology in embedded systems. The shift towards multi-core technology has revealed the demand for real-time scheduling algorithms along with synchronization protocols to support real-time applications on multiprocessors.

Two major conventional algorithms for real-time task scheduling on multiprocessors are global and partitioned scheduling. In partitioned scheduling, separate schedulers are utilized for each processor to which the tasks are statically assigned. However, in global scheduling, one global scheduler selects tasks from the unique global ready queue and tasks are allowed to migrate between different processors.

On the other hand, the semi-partitioned scheduling approach has compound both the partitioned and global techniques to create a middle approach for multiprocessor scheduling, which first has been introduced by Anderson et al. in [1]. Semi-partitioned scheduling extends partitioned scheduling by allowing a low number of tasks to migrate among different processors which has lead to an improvement of the schedulability. Similar to the partitioned scheduling approach, in semi-partitioned scheduling separate ready queues are used for each processor in which an individual scheduler manages tasks from the ready queue to have access to the processor capacity. Different task assigning techniques have been investigated in prior works [2], [3], [4], [5]. Guan et al. in [5] have increased the utilization bound of task sets as high as the utilization bound of Liu and Layland's Rate Monotonic Scheduling (RMS) for any task set.

Any scheduling approach for multiprocessors consists of three parts: (i) the partitioning method which specifies how to assign tasks to different processors, (ii) the scheduling algorithm which determines how to schedule the tasks on each processor, and (iii) the synchronization protocol which handles mutually exclusive resource sharing between tasks and is also the focus of this paper. Execution of tasks on multiprocessors can cause longer blocking delays compared to the uniprocessor case since other processors in the system can contribute in incurring delays for one specific processor. Therefore, an efficient real-time synchronization protocol is needed for accessing the shared resources under semi-partitioned scheduling. In this paper we present two solutions to handle resource requests in a shared environment under semi-partitioned scheduling.

A. Contributions

Our contributions in this paper are as follows:

- We first propose two algorithms for handling resource sharing under semi-partitioned scheduling. We formulate the system model and deploy some rules to manage requests on local and global resources along with defining the resource queue structure.
- Then we elaborate the blocking conditions and formulate the blocking terms based on the system model.
- Finally we perform experimental evaluations for both proposed algorithms to evaluate and compare the schedulability results of both algorithms under semi-partitioned scheduling.

B. Related Work

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. In the following, we will present a brief description of most related existing synchronization protocols developed for partitioned scheduling algorithms.

Rajkumar et al. proposed a synchronization protocol [6] which later was called Distributed Priority Ceiling (DPCP) [7]. DPCP is a synchronization protocol for shared memory multiprocessors. In DPCP a job executes its local and non-critical sections on its assigned processor while its global critical sections may execute on processors other than its allocated processor. Processors which execute global critical sections are called synchronization processors. There can exist more than one synchronization processor in a system, however global critical sections which request the same resources should be executed on the same synchronization processor. The protocol utilizes local agents for handling resource requests.

Later, Rajkumar et al. proposed the Multiprocessor Priority Ceiling Protocol (MPCP) in [6] which is the extension of the Priority Ceiling Protocol (PCP) [8] proposed by Sha et al. for multi-core platforms under fixed priority scheduling. In MPCP a task requesting a resource is suspended if the resource is not available at the moment. Tasks related to different processors waiting on a specific resource in the system will be enqueued in a global prioritized queue. According to MPCP the priority of a task requesting global resources is boosted within its critical section to a priority higher than any task in that processor. As a result the remote blocking duration of a job is bounded to critical sections of other jobs, which is negligible compared to tasks execution times.

Multiprocessor the Stack Resource Policy (MSRP) proposed by Gai et al. in [9] is another synchronization protocol which is the extension of Stack Resource Policy (SRP) by T. Baker [10] for multiprocessors. In MSRP when a task tries to get access to a global resource which is already locked in another processor, it is inserted to a global FIFO queue and performs busy wait which is called *spin lock*. The implementation complexity is higher for the MPCP protocol compared to the MSRP protocol while MSRP waste the capacity of the CPU more due to spin lock waiting on a global resource. At first sight it seems that MSRP performs better when the critical sections are small, and MPCP performs better when the critical sections are longer. Yet it should be considered that due to more blocking in MPCP the CPU faces more context switch overhead which is considerably high compared to MSRP.

Flexible Multiprocessor Locking Protocol (FMLP) was introduced by Block et al. in [11] for both partitioned and global scheduling algorithms. Later the partitioned FMLP was extended by Brandenburg and Anderson in [12] for fixed priority scheduling. According to FMLP, resources are divided to long and short resources while the classification of resources in terms of long or short is upon the definition of the user. Tasks blocked on long resources are suspended and wait in a FIFO queue while tasks blocked on short resources perform busy-wait. A difference of FMLP compared to other algorithms such as MPCP and MSRP is that FMLP supports nested global critical sections.

$O(m)$ Locking Protocol (OMLP) proposed by Brandenburg and Anderson in [13] is another locking protocol for handling resource sharing in multiprocessors. OMLP is a *suspension-oblivious* protocol. Under a suspension-oblivious protocol suspensions are contributed in execution time of tasks which means that suspended jobs are assumed to occupy the processor. In contrast, other suspension-based protocols are *suspension-aware* protocols in which true suspension of tasks is considered and suspended jobs are not assumed to occupy the processor. Furthermore OMLP is implied as an *asymptotically optimal* protocol. This means that the blocking duration for the whole task set in the system is confined to a fixed factor of blocking which is unavoidable in the worst case for some task sets. Tasks that compete with each other for global resources under the partitioned OMLP first have to acquire a unique token devoted to each processor through waiting in the local priority-based queue of the related processor. As soon as the task holds the token it enqueues in the global FIFO queue of the resource.

Multiprocessor Synchronization Protocol for Open Systems (MSOS) is a synchronization protocol for resource sharing among independently-developed real-time applications presented by Nemati et al. in [14]. Under MSOS applications/subsystems have been developed independently, meaning that their scheduling algorithm and priority settings may differ from each other. There is one FIFO queue for each global resource in the system in which the processors requesting the resource will be enqueued. The requests for a global resource in each processor are handled by a local queue in that processor in which tasks requesting the resource will be enqueued. The local queues are either FIFO or priority based queues.

Inspired by the previous works we have investigated two techniques for handling resource requests under semi-partitioned scheduling regardless of the partitioning algorithms. In the rest of this paper we present the proposed methods besides analysis the relevant blocking terms and evaluate the performance results.

II. System Model

In this section we introduce the system model. The multiprocessor platform consists of a task set of n periodic tasks $\{\tau_1, \tau_1, ..., \tau_n\}$ running on m processors $\{P_1, P_2, ..., P_m\}$. Each task τ_i is identified by the (C_i, T_i, ρ_i) model where C_i is the worst-case execution time, T_i is the minimum inter-arrival time between two successive jobs of task τ_i, and ρ_i is the priority of the task τ_i. Tasks in the system have implicit deadline, i.e., the relative deadline of any job of task τ_i is equal to T_i. A task τ_i is assumed to have priority higher than that of task τ_j, if $\rho_i > \rho_j$. For the ease of evaluation, we assume that each task in the system, has a unique priority.

Under the semi-partitioned approach, some tasks are assigned just to one processor and they are called non-split tasks. The tasks which can not completely fit into one processor are split and allocated to more than one processor and they are called split tasks. However each single part of a split task which is allocated to different processors is called a subtask of a split task. All subtasks of each split task are assumed as normal tasks in the system, however, they should be synchronized with each other. This means that a subtask should finish its execution prior to its successive subtasks. In other words, a subtask of a split task can not start to execute before the former subtask has completed.

In the example shown 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. The arrival time of the task τ_i is denoted by a and T_i shows τ_i's deadline. As it can be seen, τ_i^2 becomes ready to execute with a constant offset which is equal to the τ_i^1's worst-case response time denoted by r_i^1. Similarly τ_i^3 arrives with a constant offset equal to the worst-case response time of τ_i^2. Yet the deadline of all subtasks and therefore the whole task is T_i. Accordingly, we present the subtasks of split tasks except the first subtask with the (C_i, T_i, ρ_i, O_i) model, where O_i denotes the constant offset caused by the delay imposed from the former subtask's maximum response time. The priority of all subtasks belonging to a split task τ_i

is identical and the same as task τ_i's priority. This is because having various priorities assigned to different subtasks of a split task will lead to different worst case blocking durations under the first proposed algorithm for a specific resource request, since the resource may be requested in any subtask of the split task.

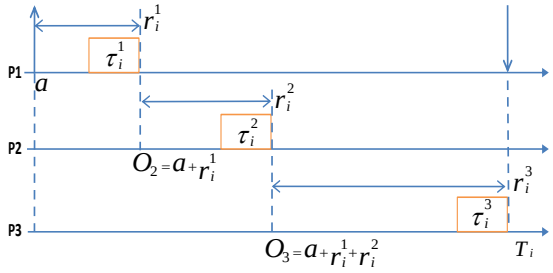

Fig. 1. Subtasks in a split task

The set of tasks on processor P_k that request a specific resource such as R_q are denoted by $\tau_{q,k}$. On the other hand, the tasks on processor P_k may share a set of resources. These resources are identified by R_{p_k} and are protected by semaphores. The shared R_{p_k} resources can be either the local or global resources. Local resources are resources which are only shared by tasks on the same processor, while global resources are resources shared by tasks on different processors. Without loss of generality, the set of local and global resources are denoted by $R^L_{P_k}$ and $R^G_{P_k}$ respectively. Moreover, $Cs_{i,q}$ denotes the worst-case execution time of the longest critical section of task τ_i in which it requests the resource R_q. Furthermore, $n^G_{i,q}$ identifies the number of global critical sections of task τ_i (gcs) in which it requests the resource R_q.

III. General Description

Under semi-partitioned scheduling, a major group of tasks is assigned to fixed processors while a low number of tasks which can not completely fit into one processor during the allocation process will be split among different processors. Similar to partitioned scheduling, in semi-partitioned scheduling, each processor has its own scheduler and ready queue in which tasks competing for the processor capacity are enqueued. However, the allocation mechanism, i.e., how to assign tasks to the processors in the system or how to split them once they can not totally fit in one processor is not investigated in this paper. Instead, our focus in this paper is how to handle 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 a proper algorithm under which tasks are assigned to processors in the sense of a semi-partitioned protocol, e.g., the approach of Guan et al. introduced in [5].

Once a processor can not dedicate enough capacity to a task, the task will be divided into subtasks. The first part of the task is assigned to the current processor until the processor capacity is filled in the sense that no more tasks can be allocated to the processor. The capacity criteria under which tasks are assigned until the processor is filled, is the utilization bound of the processor in this case. The remainder of the task which could not fit in the current processor, will be assigned to the next processor. If the remainder of the task could not fit completely in the next processor, it will be split in the same way. The subtasks of a split task which fill the processor capacity are called the body of the split task. It may happen that during the process of splitting a task into different processors, the remainder of a split task can not fill the processor. The subtask which do not fill the last processor, i.e., other tasks can still be allocated to the processor, is called the tail of a split task.

Note that, the allocation process is done before the system is scheduled. This means that, first the task set is allocated to the processors in the system and then the tasks will be scheduled during runtime on each processor.

Each processor in the system can only contain one body of any split task, since according to the notion of split tasks, the body fills the processor during the allocation process, and no other task can be allocated to the processor anymore. Therefore, if more than one subtask is allocated to a processor, at most one of them is the body of split task and all the other subtasks are tail of other split tasks.

Both split and non-split tasks may request mutually exclusive resources under semi-partitioned scheduling. The resources are guarded by semaphores to guarantee the mutual exclusive access.

A priority-based global queue is considered for each global resource in the system in which tasks requesting the related resource are enqueued. Furthermore, a priority-based local resource queue is devoted to each processor to enqueue the tasks allocated to the processor which have been granted access to different global resources. A task is granted access to a resource when the resource is available and the task locks the resource. But it may happen that a task which has locked a resource can not execute its critical section immediately. As soon as the task starts to execute its critical section, the task get access to the resource. When a task requests a resource, if the resource is not available at the moment, the task is inserted to a priority-based queue which has to wait with all other tasks from other processors that have requested the same resource. As soon as the resource is granted to the task, it is removed from the global resource queue and inserted to a local priority-based queue on its assigned processor. The task on the local resource queue has to wait for the processor time along with all other tasks of the processor which also have been granted other global resources.

In general, most computer tasks do not have a fixed execution time. The possible execution flows of a task, i.e., different execution paths in the code, cause certain variations of executions for a typical task. Variations in the execution of a task can happen in terms of different input data, and software characteristics such as loop iterations, nested loops, infeasible paths, execution frequencies of code parts, etc [15], [16], [17]. As a general model of this behavior, it can be observed that the critical sections of tasks may happen at different times within the task execution. In other words, a conservative assumption is to assume that a critical section may occur at any time during the task execution in different task instances. In the case of split tasks, this leads to the situation that the critical sections may happen in different subtasks of the split task and consequently on different processors. Therefore, there is no guarantee that a specific critical section happens in the same

subtask in different instances of the task and subsequently on the same processor. However, it should be noticed that a specific critical section in a split task can only occur in one of the subtasks in each task instance, and once it is finished the same request will not occur in any other subtask of that task instance. Note that, the same resource can be requested many times by a task, but as it is obvious, once it is requested in one of the subtasks, it can not be repeated in other subtasks. Next we will elaborate more the structure of queues which are suggested to manage the global resources under the proposed synchronization protocols.

All resources requested by subtasks of split tasks are global resources. Since a critical section of a split task may occur in any of the subtasks of the split task which are on different processors, the resources requested by split tasks are by definition global resources. Variations in execution times of a task is the reason that the same critical section may occur in different subtasks in various task instances.

A. Resource queues structure

As mentioned above, a priority-based global queue is considered for each global resource in the system, in which the tasks requesting the same resources are enqueued. The tasks on the same processor that are granted access to different global resources are served in a local priority-based queue. The tasks are enqueued in both global and local resource queues with their original priority. An example of the resource queues has been presented in Figure 2.

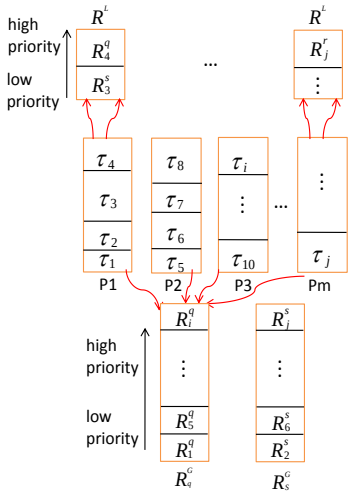

Fig. 2. System resource queues

Whenever a task requests a resource, if the resource is not available, the task is inserted to the global queue of the resource. As soon as the global resource is granted to a task, if the processor is busy with executing a task using a global resource it is inserted to the local queue of the assigned processor. A task is granted access to a resource, either when the resource is available at the time when the task requests it, or when the task has the highest priority among all tasks in the related global resource queue and the resource has been released by any other task in the system. After the task is granted access to its requested resource, it has to wait in the local resource queue of its assigned processor along with all other tasks on that processor which have requested and also granted access to other resources. However, if the task which has been granted access to a resource is the highest priority task in the local resource queue and also the processor is free, the priority of the task is boosted to the highest priority on the processor on which it should execute. Therefore, the task starts to execute its gcs non-preemptively and releases the resource once it is completed.

In the Figure 2, R_i^q denotes the request of task τ_i to R_q. As it can be seen, tasks requesting the same global resources from different processors are enqueued in the same queue. In this example, the global queue for resource R_q denoted by R_q^G, holds requests of tasks from different processors on global resource R_q. In this example, task τ_1 on processor P_1 and task τ_5 on processor P_2 both have requested resource R_q and are enqueued in R_q's related global queue since the resource was not available at the time (task τ_4 on processor P_1 has already locked it). The same situation has been shown for tasks τ_2 and τ_6 from processors P_1 and P_2 respectively, that have requested the global resource R_s which already has been locked by task τ_3 from processor P_1.

Local queues R_1^L and R_2^L in this example are local queues of processors P_1 and P_2 respectively, which enqueues requests of tasks in P_1 and P_2 on different global resources. As it can be seen in Figure 2, task τ_3 which has requested R_s and task τ_4 which has requested R_q on processor P_1, are waiting in the R_1^L queue since the resources R_s and R_q have been granted to them but the processor was not yet free.

Example. An example of global resource handling is depicted in Figure 3. As shown, P_1, P_2 and P_3 are processors in the system to which the tasks τ_1 to τ_8 are allocated. The tasks τ_2, τ_3 and τ_8 are assigned to processor P_1, while the tasks τ_4, τ_5, τ_6 and τ_7 are assigned to processor P_2 and τ_1 belongs to processor P_3. As mentioned, subtasks of split tasks behave as normal tasks in the system, although each subtask has a constant offset according to its previous subtask. The period of all tasks in this example is 20. Consider that, two subtasks of a split task can not execute simultaneously, therefore during one period which has been depicted in this example none of the tasks belong to one split task. There are three global resources R_1, R_2 and R_3 which are shared among the tasks on three processors. In this example a task τ_i has a priority higher than that of τ_j, if $i > j$, e.g., τ_7 has a priority higher than that of τ_6.

- At time 1 τ_1 requests R_1 and as it is available, τ_1 locks the resource. As processor P_3 is free, τ_1 starts using its resource. Therefore, when at times 2, 5 and 6 the tasks τ_3, τ_4 and τ_6 respectively request R_1, they are all blocked on R_1 and are inserted into R_1's global queue.
- At time 3 τ_2 requests R_2 and since it is available, it starts using R_2. Therefore, when at time 4 τ_5 requests R_2, it is blocked and therefore inserted to R_2's global

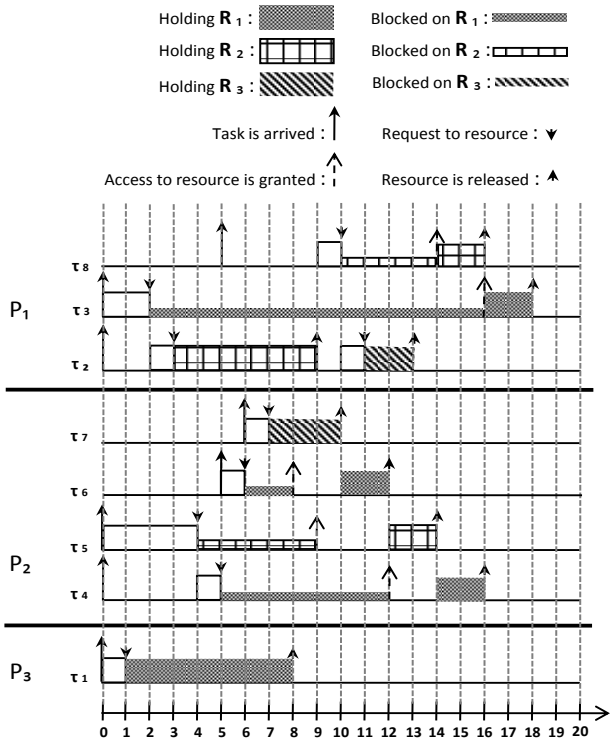

Fig. 3. Resource requests handling

queue. Notice that, τ_8 arrives at time 5 and although it is the highest priority task on P_1, it has to wait until τ_2 completes executing its gcs (with the boosted priority) until time 9.

- τ_7 requests R_3 at time 7 and since it is available, it starts to execute its gcs using R_3. As a result, τ_5 and τ_6 which are granted access to their requested resources, since τ_1 releases R_1 at time 8 and τ_2 releases R_2 at time 9, have to wait in P_2's local resource queue until τ_7 finishes its gcs (with the boosted priority on P_2). When τ_7 completes its gcs at time 10, τ_6 starts to execute using R_1 since it has a higher priority than that of τ_5 in the local resource queue of P_2.
- τ_8 requests R_2 at time 10 and is blocked since τ_5 has already been granted access to R_2 at time 9. τ_8 waits until τ_5 releases R_2 at time 14 and then starts to execute its gcs.
- As soon as τ_6 is completed, τ_4 is granted access to R_1, but has to wait until time 14 for the higher priority task τ_5 to complete its gcs.
- When τ_4 releases R_1 at time 16, R_1 is granted to τ_3 immediately since no higher priority task has requested it. τ_4 blocks τ_8 at time 16 as it has exited its gcs, and starts to execute its gcs (with the boosted priority).

B. MLPS

Migration-based Locking Protocol under Semi-partitioned scheduling (MLPS) is the first proposed algorithm which is based on centralizing critical sections of each split task into one marked processor. The marked processor is the processor on which always all critical sections of the original task will be executed. For each split task in the system one marked processor is defined on which all resource requests in any subtask of the related split task is served. Under MLPS, every time a job in a subtask of a split task requests a resource, it releases its allocated processor (source) and migrates to the marked processor (destination). The migrated task then executes its critical section non-preemptively on the marked processor. Once the job finishes its critical section on the marked processor, it will migrate back to its original processor.

Note that, two different kinds of migration can occur related to the split tasks in a semi-partitioned protocol. One type of migration is the migration of each split task from a subtask to the next subtask on another processor. This migration happens at specific time slots for each split task. These specific points in time are distinguished by the worst case response time of each subtask. At the point of implementation, these fixed points can be specified by using timers in the system. The second type of migration is the migration happening under MLPS, in which a subtask of a split task migrates to its marked processor to execute its critical section.

Note that, if the migration to the next subtask on a marked processor occurs within a critical section, an overrun will happen on the marked processor, i.e., the task will continue to execute its gcs. In other words, the task stays on the processor until it finishes execution of its gcs, and then migrates to the next processor. This will prevent extra migrations under this situation. This will not cause any problem for the schedulability of the other tasks on the processor, since the marked processor by the definition mentioned previously, can fit all critical sections of a split task. The migration to the next subtask in a split task within a critical section do not occur on processors other than the marked processor, since the tasks migrate once they request a resource.

Next, we will clarify the MLPS protocol in terms of a set of rules which is discussed as follows:

Rule 1: Access to local resources in both algorithms is handled by a uniprocessor synchronization protocol, e.g. PCP or SRP.

Rule 2: Tasks execute their gcs with the boosted priority. If ρ_h is the highest priority of any task in processor P_r on which a task τ_i is supposed to execute its gcs, then τ_i's boosted priority will be $\rho_h + 1$ within its critical section. This priority boosting allows the task to execute its critical section non-preemptively. When the task exits its gcs, its priority is changed back to its original priority.

Rule 3: When a task requests a global resource, if the resource is not available at the moment, the task is suspended, and it is enqueued in the resource's global queue. The task is inserted to the resource's global queue with its original priority. As soon as the task becomes the highest priority task in the queue, the resource is granted to the task and the task is removed form the queue.

Rule 4: When a non-split task is granted access to a global resource, if the processor to which it is allocated, is not idle at the moment, the task is inserted to the processor's local queue with its original priority.

Rule 5: As soon as any task is in the head of the local resource queue of a processor, i.e., it is the highest priority task in the queue, its priority is boosted to a priority higher than any

priority on that processor (boosted priority).

Rule 6: When a subtask located on a processor other than its marked processor requests a global resource, it is migrated to the marked processor. Once the resource is granted to the subtask, it is inserted to the local queue on the marked processor if the processor is not free.

Rule 7: Resource requests of a subtask on its marked processor are served on the same marked processor and no migration happens in this case. The resource requests are in this case handled similar to non-split tasks mentioned in Rule 4.

Rule 8: When a task releases a resource the resource becomes available to the next highest priority task in the global queue of the resource, if any. The subtask that migrated to the marked processor will migrate back to its original processor as soon as it finishes its critical section.

Note that, requests of the non-split tasks in the system are served on the processor they are allocated.

As a result of extra migrations under MLPS, the subtasks will incur overhead mainly due to cache-related migration overhead [18]. This overhead is caused by additional cache misses that a job incurs when it resumes after migrating to the marked processor.

As it can be seen in Figure 4 subtask τ_i^2 requesting a resource in processor P_2 migrates to processor P_1 which is its marked processor and it migrates back to its original (assigned) processor P_2 after executing its critical section. As shown in the figure, two migration overheads can occur during the execution of τ_i^2.

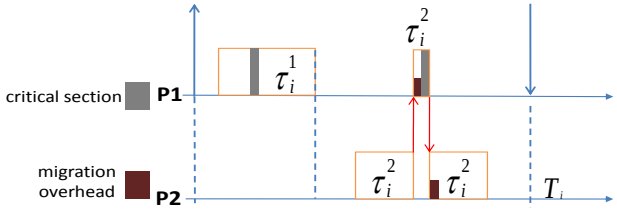

Fig. 4. Resource handling in migration-based synchronization protocol

C. NMLPS

Non Migration-based Locking Protocol under Semi-partitioned scheduling (NMLPS), is the second proposed algorithm in which the resource requests by split tasks will be served on the same processor where the requests occur. As a result, no extra migration happens under NMLPS. Rules 1, 2, 3, 4 and 5 in MLPS are also applicable for NMLPS, so we will continue with Rule 6.

- **Rule 6:** If a subtask on any processor is granted access to a resource, if the processor to which it is assigned to is not idle at the moment, it is inserted to the resource local queue of the processor with its original priority. A subtask will start executing its critical section with the boosted priority as soon as it is the highest priority task in the queue.
- **Rule 7:** The task releases the resource once it completes its critical section, and the resource becomes available for the next highest priority task in the global queue of the resource, if any.

Under NMLPS, since subtasks are assumed to access global resources in their original processors, they introduce a delay to the local tasks due to the access of global resources which is not the case in MLPS. As shown in the example in Figure 5, subtask τ_i^2 executes its critical section on its original processor P_2. Please notice that at any time only one subtask of a split task can be located in a global queue.

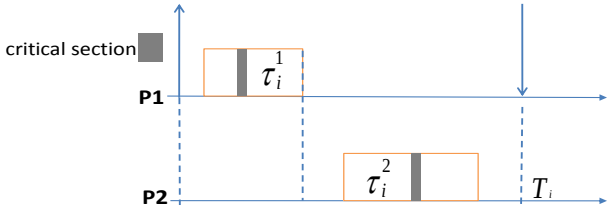

Fig. 5. Resource handling in non-migration-based synchronization protocol

A task requesting a global resource may also be blocked by tasks on other processors, which is called remote blocking. Accordingly, the processors causing this blocking are called remote processors. Under NMLPS, all subtasks of a split task are assumed to share resources with other tasks that share resources with the original task. Therefore the number of tasks which cause remote blocking to other tasks in the system are increased under NMLPS. In contrast, under MLPS, by centralizing critical sections of split tasks into one processor the number of tasks that cause remote blocking is decreased by this approach. However, we can not ignore the fact that overhead is increased under MLPS due to migration of the subtasks to the marked processors.

IV. Schedulability Analysis

In this section we present the schedulability analysis of the two proposed algorithms. A task is said to be blocked on a resource when the resource is not available for the task at the moment it requests the resource. There are various possible situations which may cause a task to be blocked on resources. Next, we will enumerate four possible scenarios in which a task may experience blocking in a multiprocessor platform under semi-partitioned scheduling. We categorize the blocking terms into local and remote blocking. When the blocking is imposed by local tasks, i.e., the tasks executing on the same processor, it is called local blocking. On the other hand, the blocking caused due to tasks on remote processors is called remote blocking.

A. Local blocking due to local resources

We denote n_i^G as the number of global critical sections that any job of τ_i executes before its completion. Each time a task τ_i is suspended on a global resource, it gives a chance to a lower priority task τ_j to lock a local resource with a ceiling higher than τ_i's priority. τ_j can then block τ_i in its non-gcs, (after it resumes and completes its gcs), e.g. the local resource handling under PCP). This kind of blocking can happen each time τ_i requests a global resource. In addition, 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 earlier than τ_i which leads to $n_i^G + 1$ times. On the other hand, τ_j can release a maximum number of $\left\lceil \frac{T_i}{T_j} \right\rceil + 1$ jobs before τ_i is finished. In addition, each job of τ_j 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. This specifies that at most $(\left\lceil \frac{T_i}{T_j} \right\rceil + 1) n_j^L(\tau_i)$ times each lower priority task can introduce this type of blocking to τ_i. Thus the first blocking term denoted by $B_{i,1}$ is as follows:

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

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 request another global resource. As a result τ_i will be blocked by τ_j later when it is resumed. This situation can be seen in Figure 3. When τ_5 is blocked on R_2 at time 4, the lower priority task τ_4 requests another resource (R_1) and later at time 14 blocks τ_5 after it has finished its gcs.

τ_i may experience this kind of blocking up to $n_i^G + 1$ times due to all its global resource requests besides the situation in which τ_j has arrived and requested a global resource before τ_i arrives. τ_j can release a maximum number of $\left\lceil \frac{T_i}{T_j} \right\rceil + 1$ jobs before τ_i is finished. However, each job of τ_j can preempt τ_i's current job at most up to n_j^G times. The blocking introduced under this terms, denoted by $B_{i,2}$, is calculated as follows:

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

C. Remote blocking

Whenever a task has to wait for a global resource which is locked by another task on other processors, it incurs a remote blocking. In our proposed algorithms under two situation tasks may experience remote blocking, which we discuss next.

1) Tasks with lower priority: A task τ_i on a processor P_k that requests a global resource R_q may get blocked since a lower priority task τ_j on another processor P_r already has been granted access R_q. In this case τ_i has to wait until τ_j releases the resource. τ_i can be blocked by at most one lower priority task τ_j on other processors which requests a resource R_q. This is because, as soon as the resource is released by τ_j, no other lower priority task than τ_i can be granted access to R_q, and if there is no other higher priority tasks than τ_i in R_q's global queue, τ_i locks it.

On the other hand, τ_j which is granted access to R_q may also wait in the local resource queue of P_r, if processor P_r is not free. Regardless of which of the proposed algorithms is used, τ_i has to wait in the local resource queue of P_r along with all other tasks which have been granted access to other resources. As soon as P_r becomes free and τ_j is the highest priority task in the local resource queue, it will get access to R_q and start to execute its gcs. In the worst-case τ_j has to wait for all higher priority tasks in P_r which are granted access to resources other than R_q. As a result, these tasks with higher priority than τ_j also contribute in blocking of task τ_i by indirectly delaying τ_i on R_q.

This kind of blocking can be seen in the example in Figure 3 at time 10, when τ_8 requests R_2 but is blocked since the lower priority task τ_5 has already been granted access to R_2 at time 9. τ_8 is also indirectly delayed by the higher priority task τ_6 than that of τ_5 on P_2 which finishes its gcs at time 12 and let τ_5 to start its gcs and release R_2 at time 14.

In order to calculate the worst-case delay introduced 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 each time τ_i requests a global resource, therefore the related blocking term which is denoted by $B_{i,3}$ is as follows:

$$B_{i,3} = \sum_{\substack{\forall R_q \in R_{P_k}^G \\ \wedge\, \tau_i \in \tau_{q,k}}} n_{i,q}^G RWRL_{q,k'} \quad (3)$$

where $n_{i,q}^G$ is the number of τ_i's global critical sections in which it requests R_q and $RWRL_{q,k'}$ is the waiting time for resource R_q on remote processors of processor k due to tasks with lower priority than τ_i and is calculated as follows:

$$RWRL_{q,k'} = \max_{\substack{\forall \rho_j < \rho_i \\ \wedge\, \tau_j \in \tau_{q,r} \\ \wedge\, k \neq r}} \{Cs_{j,q} + wlh_{j,q}\} \quad (4)$$

where $wlh_{j,q}$ is the waiting time introduced by τ_j's local higher priority tasks on the τ_i's remote processor P_r. In other words, this is the amount of time which τ_j has to wait before it can execute its gcs and release R_q.

$$wlh_{j,q} = \rho_{h,j}(R'_q) \max_{\substack{\tau_t \in P_r \\ \wedge\, \rho_t > \rho_j \\ \wedge\, R_s \in R_r^G \\ \wedge\, s \neq q}} \{Cs_{t,s}\} \quad (5)$$

where $\rho_{h,j}(R'_q)$ is the number of local tasks with priority higher than τ_j that share global resources other than R_q.

2) Tasks with higher priority: A task τ_i assigned to processor P_k which gets blocked on R_q has to wait for all higher priority tasks in the R_q's global queue to release R_q. These tasks may repeatedly block τ_i before it is granted access to R_q. τ_i incurs remote blocking from the higher priority tasks in the R_q's global queue which are located on processors other than P_k. On the other hand, a higher priority task τ_t assigned to processor P_r may also have to wait in the P_r's local resource queue for at most one critical section per each with priority higher than τ_t that requests resources other than R_q. In the worst-case τ_t has to wait for all higher priority tasks in P_r which are granted access to resources other than R_q. As a result, these tasks with higher priority than τ_t also contribute in blocking task τ_i by indirectly delay τ_t on R_q.

This situation can be seen in the example shown in Figure 3, where τ_3 which has requested R_1 sooner than τ_4, has to wait also for global critical sections of higher priority tasks τ_6 and τ_5 than that of τ_4 on $\P_2$, besides waiting for τ_4 gcs.

On the other hand, similar to the term in Section IV-C1, τ_t may generate up to $\left\lceil \frac{T_i}{T_t} \right\rceil + 1$ jobs 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 also block τ_i's current job up to $n_{t,q}^G$ times. The

related blocking term which is denoted as $B_{i,4}$ is calculated as follows:

$$B_{i,4} = \sum_{\substack{\forall R_q \in R^G_{P_k} \\ \wedge\, \tau_i \in \tau_{q,k}}} RWRH_{q,k',i} \tag{6}$$

where $RWRH_{q,k',i}$ is the maximum blocking time on R_q introduced to τ_i by remote tasks with priority higher than τ_i.

$$RWRH_{q,k',i} = \sum_{\substack{\forall \rho_t > \rho_i \\ \wedge\, \tau_t \in \tau_{q,r} \\ \wedge\, r \neq k}} n^G_{t,q}(\lceil T_i/T_t \rceil + 1)(Cs_t, q\ + wlh_{t,q}) \tag{7}$$

The total blocking of a task τ_i in the system is then calculated as follows:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} \tag{8}$$

V. Migration overhead

As mentioned before, under MLPS, the critical sections of split tasks will migrate to a specific processor (marked processor). As a result of these migrations an overhead is introduced due to migration delay. The execution time of the task which is migrated is inflated by per migration overhead twice. The migrated task incurs a delay once when it migrates to the marked processor and once it migrates back to its original processor as it can be seen in Figure 4. However, the migrated critical section is also inflated with one migration overhead itself. This is because, when a task is delayed due to a migrated task executing its gcs, besides the critical section length, it also incurs a delay caused by the migration overhead. Therefore, an overhead is also included in the critical section length.

$$C_{i_{migrated}} = C_i + C_{overhead} \tag{9}$$

$$Cs_{i_{migrated},q} = Cs_{i,q} + C_{overhead} \tag{10}$$

VI. Evaluation

In this section we present our experimental evaluation for comparison of the performance of MLPS and NMLPS. In our experiments, we compared the performance of the proposed algorithms in terms of schedulability. We generated a set of platforms each representing a multiprocessor system. Experiments are based on generating random number of processors in each platform. The assigning mechanism of allocating tasks to processors in the generated platform for both algorithms is based on first-fit partitioning.

The partitioning algorithm in a platform selects a random processor and fills it up to its assigned maximum utilization capacity. As soon as the processor is filled, the next processor is selected for assigning new tasks. If a task can not fit in the processor completely, it is split in two subtasks. The first subtask fills the current processor capacity and the next subtask will be assigned to the next processor under same procedure. The platform partitioning scheme can be seen in Figure 6.

In order to compare both algorithms schedulibility performance under identical conditions, we first generate a platform for MLPS and evaluate the schedulibility of the platform. Afterwards, the platform is changed upon NMLPS and again the schedulability is checked. In this way, the resource allocation to different tasks in the platform is the same under both algorithms and the comparison is valid.

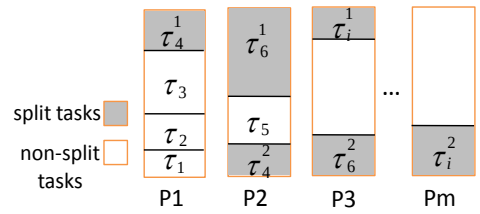

Fig. 6. Assigning mechanism based on first-fit partitioning

The experimental results show that the migration overheads in MLPS has a greater effect in making the platforms unschedulable. With large overhead values, generally the platforms become more unschedulable under MLPS. This is an important factor which makes the MLPS approach more suitable for multiprocessor platforms with tasks having small working set size *(WSS)* [18].

A. Experimental setup

In our experiments we have employed the same resource allocation and setups to evaluate the schedulability performance of both algorithms under the same circumstances. We have randomly generated 1,000,000 platforms. The number of processors is randomly selected through a specified range {4, 8, 12, 16}. The maximum utilization capacity of all processors in the platform is identical and selected through a set of predefined values {0.3, 0.4, 0.5, 0.6}. Furthermore, the number of resources which are shared among tasks in a platform is constant in each generated platform, and is 10. On the other hand, the number of critical sections created in each task are selected randomly through a set of specified values {1, 2, 3, 4, 5, 6}. The critical sections length is also selected randomly via the range {5, 25, 45, 65, 85, 105, 125, 145, 165, 185, 205}.

The migration overhead which is mostly the result of cache-related delay has been chosen randomly for MLPS algorithm from a set of values {0, 20, 60,140, 300, 620, 1260, 2540}, providing small as well as large amounts. The small overhead values has been included in the setup in order to compare MLPS with negligible overhead amounts against the performance of NMLPS.

The experiments have been applied for 1,000,000 platforms and repeated more than 3 times which each time yielded close to the same results. This indicates that this 1,000,000 samples can be representative.

B. Results

In this section we present the evaluation results of our experiments for both algorithms. The Figures depict schedulibility factor versus different parameters such as critical sections length, the number of critical section per task, the number of processors and the utilization capacity of each processor,

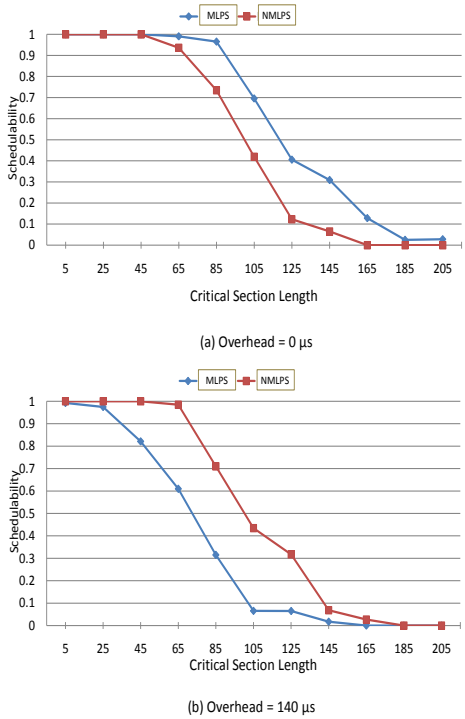

Fig. 7. Performance of synchronization protocols as the critical section lengths increases. Number of processors = 8, Processor maximum utilization capacity = 0.4, Number of critical sections per task = 3

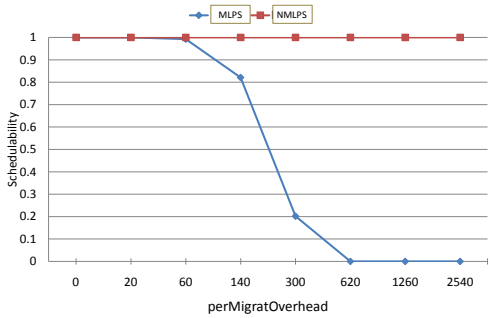

Fig. 8. Performance of synchronization protocols as overhead increases. Number of processors = 8, Critical sections Length = 45, Number of critical sections per task = 3, Processor maximum utilization capacity = 0.4

and the value of per-migration overhead. The results show that the migration overhead play a prominent role in degrading the performance of MLPS. On the other hand, MLPS performs better under negligible amount of overhead.

The results shown in Figures 7, 9, 8, 10, 11 shows that there are some important factors which affects schedulability of the platforms considerably. These factors are the length of critical sections, number of resources requested by tasks, number of processors in the system, and the utilization capacity specified for the processors. However, considering the migration overhead, the schedulibility is affected negatively by imposing the overhead under MLPS algorithm. The schedulability under MLPS versus the migration overhead has been presented in Figure 8.

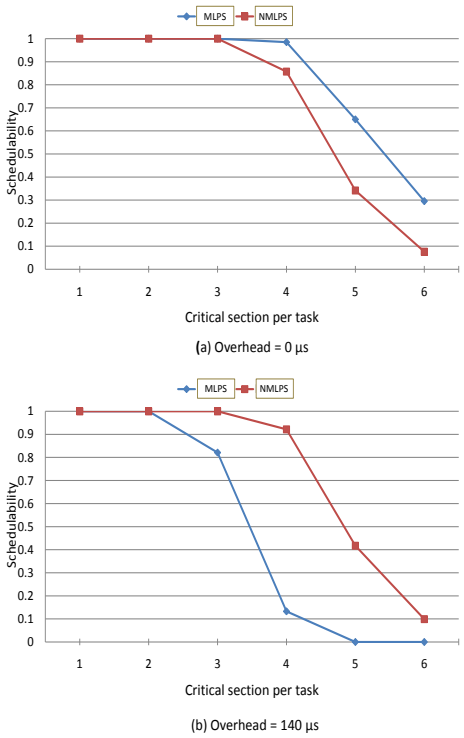

Fig. 9. Performance of synchronization protocols as the number of critical sections per task increases. Number of processors = 8, Processor maximum utilization capacity = 0.4, Critical section length = 45

The results show that MLPS suffers from migration overhead which leads to lower schedulability. However ignoring the overhead incurred by migration in MLPS, the performance of MLPS is better compared to NMLPS. This confirms the better handling of resources by concentrating the critical sections of split tasks under MLPS algorithm. This makes MLPS suitable for platforms with low migration overhead.

However, for systems with high migration overhead, NMLPS performs better, although the parameters discussed above have a prominent effect on the platform schedulability.

VII. Conclusions and Future Works

In this paper, we proposed and evaluated two synchronization protocols under semi-partitioned scheduling for multiprocessor platforms. The protocols handle the resource sharing between tasks among different processors in the platform, where some tasks have been allocated to more than one processor. The first algorithm centralizes all resource requests of the split tasks to one specific processor while in the second proposed algorithm, the requests are served on the processors they occur. We have performed experimental evaluation of our proposed algorithms in terms of comparing the performance of both algorithms against each other. The results shows that the first algorithm performs better in terms of scheduling a set of random task compared to the second protocol in presence

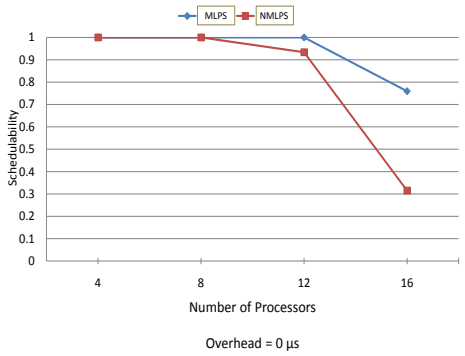

Fig. 10. Performance of synchronization protocols as the number of processors increases. Number of critical sections per task = 3, Processor maximum utilization capacity = 0.4, Critical section length = 45

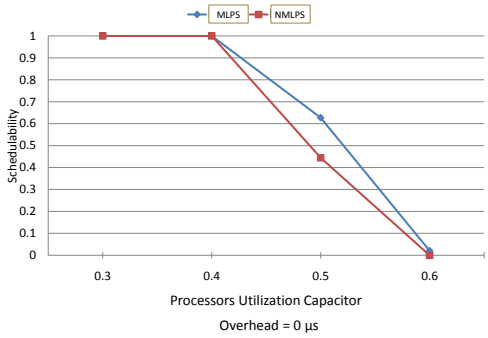

Fig. 11. Performance of synchronization protocols as the maximum processor utilization capacity increases. Number of processors = 8, Number of critical sections per task = 3, Critical section length = 45

of low amounts of overhead. However, the first algorithm impose overheads to the system due to extra migrations, which degrades it performance compared to the second algorithm.

In the future we plan to work on a new allocation mechanism that takes the presence of resource sharing in the system into consideration, to deliver a complete solution. Another interesting area of extension to this work is changing the first algorithm in such a way that the subtasks do not have to wait for a constant amount of time to migrate to their next subtasks, which is a pessimistic solution for soft real-time applications. Therefore instead of constant offsets for subtasks of a split task, a variant jitter is introduced, which should be considered in the analysis, as well.

REFERENCES

[1] J. Anderson, V. Bud, and U. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, july 2005, pp. 199 – 208.

[2] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –12.

[3] K. S. and Y. N., "Semi-partitioned fixed-priority scheduling on multiprocessors," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, april 2009, pp. 23 ,32.

[4] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, july 2009, pp. 239 –248.

[5] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-priority multiprocessor scheduling with liu and layland's utilization bound," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, april 2010, pp. 165 –174.

[6] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symposium, 1988., Proceedings.*, dec 1988, pp. 259 –269.

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

[8] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175 –1185, sep 1990.

[9] 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 *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, may 2003, pp. 189 – 198.

[10] T. Baker, "Stack-based scheduling of real-time processes," *Journal of Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[11] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, aug. 2007, pp. 47 –56.

[12] 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.

[13] B. B. Br and J. H. Anderson, "Optimality results for multiprocessor real-time locking."

[14] F. Nemati, M. Behnam, and T. Nolte, "Independently-developed real-time systems on multi-cores with shared resources," in *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, july 2011, pp. 251 –261.

[15] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Real-Time in Sweden (RTiS) 2007*, August 2007. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1323

[16] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis," in *Seventh International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, July 2007. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1317

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[18] A. Bastoni, B. Brandenburg, and J. Anderson, "Is semi-partitioned scheduling practical?" in *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, july 2011, pp. 125 –135.