MÄLARDALEN UNIVERSITY
SWEDEN

On Resource Sharing under Multiprocessor Semi-Partitioned
Scheduling

Master Thesis

Sara Zargari Afshar

April 2012

Supervisor: Farhang Nemati
Examiner: Thomas Nolte

Abstract

Semi-partitioned scheduling has been the subject of interest compared to conventional global and partitioned scheduling algorithms for multiprocessors due to better utilization results. In Semi-partitioned scheduling most of tasks are assigned to fixed processors while a few number of tasks are split up and allocated to different processors.

Various techniques have been proposed recently on different assigning protocols under semi-partitioned scheduling. Yet an appropriate synchronization mechanism for resource sharing in semi-partitioned scheduling have not been investigated. In this thesis we propose two methods for handling resource sharing under semi-partitioned scheduling on multiprocessor platforms. The main challenge is handling the resource requests of tasks that are split over multiple processors.

The solutions include handling non-split tasks as well as split tasks over requests for shared resources in the system. In this thesis we investigate delays caused by blocking on resources. Furthermore, we perform the schedulability analysis for both algorithms.

Finally we evaluate the performance of our proposed synchronization algorithms by means of experimental evaluations.

Acknowledgement

I would like to thank my supervisor Farhang Nemati for his recommendations and guidance during this thesis work. I am very thankful for the nice group work and discussions we had during this thesis project. Moreover, I would like to thank Thomas Nolte who helped and supported me in publishing scientific paper from this thesis work. I am grateful for their comments and feedbacks on my work. At the end, I would like to take this opportunity to thank my lovely parents and family besides my kind husband Mohammad, who always have supported me in my life.

Contents

1	Introduction	1
1.1	Terminology	1
1.2	Related works	3
1.2.1	DPCP Synchronization Protocol	3
1.2.2	MPCP Synchronization Protocol	3
1.2.3	MSRP Synchronization Protocol	4
1.2.4	FMLP Synchronization Protocol	5
1.2.5	OMLP Synchronization Protocol	6
1.2.6	MSOS Synchronization Protocol	7
2	Theoretical Background	9
2.1	Global Scheduling	9
2.2	Partitioned Scheduling	9
2.3	Semi-Partitioned Scheduling	11
3	Thesis Goal	13
3.1	Aim of thesis	13
3.2	Assumptions	13
4	System Model	14
4.1	Task Set Model	14
4.2	Resource Sharing	15
5	Algorithm Description	15
5.1	Resource queues structure	17
5.2	Migration-based Synchronization Protocol	19
5.3	Non-Migration-based Synchronization Protocol	20
6	Schedubility Analysis	21
6.1	Local Blocking due to Local Resources	21
6.2	Local Blocking due to Global Resources	22
6.3	Remote Blocking	22
6.3.1	Tasks with Lower Priority	22
6.3.2	Tasks with Higher Priority	23
7	Migration overhead	24
8	Remarks on blocking terms	24
8.1	Local Blocking due to Local Resources	24
8.2	Local Blocking due to Global Resources	25

8.3 Remote Blocking	26
9 Response Time Analysis	26
10 Evaluation	27
10.1 Experimental setup	28
10.2 Results	29
11 Conclusions and Future Works	32
11.1 Summary and Conclusion	32
11.2 Future works	36

1 Introduction

Research studies on real-time scheduling techniques suitable for multiprocessor systems has largely increased due to dramatically rise of interest towards multi-core systems in the marketplace. The shift towards multi-core platforms 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 references [8, 9]. In partitioned scheduling approach each task is statically assigned to a single processor on which all of its jobs will execute. A separate local ready queue and scheduler is required to independently schedule the tasks in each processor. In global scheduling approach a global ready queue is used to store all ready tasks in the system. The global scheduler then selects the highest priority tasks among all ready tasks in the ready queue for executing on the available processors.

Semi-partitioned scheduling is an overlap between pure partitioned and global scheduling approaches which was first introduced by Anderson et al. in [1]. Semi-partitioned scheduling extends partitioned scheduling by allowing a small number of tasks to be split among different processors thereby improves the schedulability, while other tasks in the system are allocated to fixed processors. Similar to partitioned scheduling, a semi-partitioned algorithm utilizes separate ready queues for each processor. 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 [16, 17, 19, 14]. Guan et al. in [14] 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: (i) the partitioning algorithm which determines how to allocate tasks or split them if required among processors, (ii) the scheduling algorithm which specifies how to schedule the tasks assigned to each processor and (iii) synchronization protocol to manage resource sharing, which is the focus of this thesis. Executions of tasks on multiprocessors may cause longer blocking delays compared to the execution of tasks on uniprocessors, due to delays caused by other processors in the system. The need of a real-time synchronization protocol for accessing the shared resources is magnified under semi-partitioned scheduling. For this purpose two algorithms have been presented in this report for handling shared resources in semi-partitioned scheduling.

1.1 Terminology

In this section we will describe some terminologies which will be used frequently in this report.

- **Global resource:** a resource which is requested and shared by different tasks on different processors on a multiprocessor platform.
- **Local resource:** a resource that only the tasks on the same processor request and share.
- **Global resource queue:** is a queue in which all tasks from different processors which request the same resource are enqueued. The structure of the global queue can be priority-based or FIFO-based (First In First Out). In the priority-based queue, tasks are ordered in the queue according to their priority and the highest priority task in the queue will be granted the access to the resource. However, in a FIFO-based queue tasks are ordered based on their request turn. The earlier a task has requested a resource the sooner it will be granted access to the resource.
- **Granting access versus having access to a resource:** the access to a resource is granted to a task when the task can have access to the resource. However, the task which has been granted access to a resource, may not start executing immediately. As soon as the task starts to execute its critical section in the processor, it means that the task has access to the resource and has locked the resource. When the task completes its critical section, it releases the resource.
- **Priority inversion:** the phenomenon in which a task is blocked by lower priority tasks for an unbounded amount of time.
- **CPU Utilization:** is a measure of real-time system performance. The CPU continues to fetch, decode, and execute instructions as long as the power is applied. The measure of the time spent doing idle processing, indicates how much real-time processing is occurring. The (CPU) utilization is a measure of the percentage of non-idle processing.
- **WSS:** represents the amount of memory occupied and needed time for handling the task's jobs.
- **Critical section:** a critical section is a sequence of statements in the code that must appear to be executed indivisibly (or atomically).
- **Mutual exclusion:** means that only one task is using the resource at a time.
- **semaphore:** a semaphore is a data structure used for protection of critical sections. In more general terms, we can say that semaphores are used for synchronization between tasks by providing mutual exclusion when several tasks are accessing the same resources.

1.2 Related works

Often, tasks execute asynchronously, i.e., at different speeds, but they may need to interact with each other, e.g., to transfer data to each other, or to access shared resources. Most real-time operating systems present different techniques for handling task communications. These techniques are required when the number of tasks become large in the system, otherwise tasks interfere with each other and may corrupt information.

Therefore, an appropriate technique is needed to handle tasks communications which is called synchronization protocol. A synchronization protocol handles the resource requests of different tasks in the system. The goal of a synchronization protocol is to bound waiting times of tasks which share resources in the system. Vast amount of work has been done on resource sharing synchronization mechanism under partitioned scheduling algorithms.

1.2.1 DPCP Synchronization Protocol

Rajkumar et al. proposed a synchronization protocol [25] which later was called Distributed Priority Ceiling (DPCP) [24]. DPCP is the extension of Priority Ceiling Protocol (PCP) [27] proposed by Sha et al. 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 same resources should be executed on the same synchronization processor. The protocol utilizes local agents for handling resource requests.

A local agent A_i^q laid on a synchronization processor, which is a processor other than task τ_i 's assigned processor q , serves the requests of τ_i jobs. On the synchronization processor a job τ_i^j of task τ_i sends a request for a global resource l to the A_i^q and suspends. Then, A_i^q executes the critical section with a priority higher than any normal task on the related synchronization processor. τ_i^j resumes when A_i^q has completed the request. When A_i^q is executing the τ_i 's request of global resources, the τ_i 's assigned processor is free and other tasks can have access to the processor.

1.2.2 MPCP Synchronization Protocol

Rajkumar et al. proposed multiprocessor priority ceiling protocol (MPCP) [23] which is the extension of PCP for multiprocessor platforms under fixed priority scheduling. The major difference between DPCP and MPCP is that in MPCP global requests are served on the same processor that the requesting task is assigned to and global resources are not assigned to any particular processor like in DPCP. Therefore local agents are not required

since jobs execute their critical sections on their allocated processor.

In MPCP the task requesting a resource is suspended if the resource is not available at the moment. Tasks allocated to different processors waiting for a specific resource will be enqueued in a global prioritized queue. According to MPCP the priority of a task requesting global resources is boosted withing its critical section to a priority higher than any task in that processor which is called *remote ceiling*. A global critical section can be preempted by higher priority global critical sections, only. As a result the remote blocking duration of a job is bounded to critical sections of other jobs which is short compared to tasks execution time within non-critical sections. Nested critical sections are not supported in the MPCP algorithm.

When a job requests a global resource the resource is granted to the job if it is not currently held by another job. If a request for a global resource cannot be granted then the job is added to a prioritized queue related to that resource. The priority used for queuing in the global resource queue is the original priority assigned to the job. When a job releases a global resource the highest priority job in the related resource queue will be eligible for locking the semaphore.

1.2.3 MSRP Synchronization Protocol

Multiprocessor Stack Resource Policy (MSRP) [13] introduced by Gai et al. [12] is another resource sharing technique 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 spin lock. The implementation complexity is higher in MPCP protocol compared to MSRP, however, MSRP wastes the capacity of the CPU more, because of spin lock on global resources. At first sight it seems that MSRP performs better when the critical sections are small, and MPCP works better when the critical sections are longer [12]. However, it should be considered that due to suspension in the MPCP algorithm, more context switch overhead is introduced which is considerably high compared to MSRP. Nested critical sections are not supported in the MSRP.

When a task tries to get access to a global resource which is already locked in another processor, it performs a busy wait. The spin lock time should be minimized, which means that the global resource should be released as soon as possible. As a result, the tasks become non-preemptable when executing a critical section on a global resource.

When a task bound to a processor tries to have access to a global resource it checks if the resource is free or not. If the resource is free then it locks it and executes its critical section. Otherwise, the task is inserted to a FIFO queue on the global resource

and performs a busy wait. When a task releases a global resource the algorithm checks the related FIFO resource queue and picks the first task to grant the access to the global resource to. If there is no more task in the queue, the global resource is released and the task becomes preemptable again.

1.2.4 FMLP Synchronization Protocol

The Flexible Multiprocessor Locking Protocol (FMLP) was proposed by Block et al. in [4] for partitioned and global scheduling algorithms. Later, the partitioned FMLP was extended by Brandenburg and Anderson in [6] for fixed priority scheduling approach. The term flexible refers to the fact that it can be adapted for use under both global and partitioning scheduling.

According to FMLP resources are divided into long and short resources while the classification of resources in term of long or short is upon the user definition. Tasks which get blocked on long resources are suspended and wait in FIFO queue while tasks get blocked on short resources perform busy-wait. In contrast to MPCP and MSRP protocols, FMLP supports nested global critical sections.

In FMLP the priority of the jobs holding global resources are boosted to the highest priority in the system in order to be able to preempt jobs executing in their non-critical sections. If there are more than one job with boosted priority, they will be served in a FIFO queue. The priority boosting is not needed in case of short resources as they are performing a non-preemptable spin lock. Both short and long requests are executing non-preemptively.

Nested resource requests may cause deadlock and have negative effect on waiting durations of tasks for resources. The FMLP supports nesting resources by organizing resources in *source groups*. Source groups are sets of resources which has been classified in terms of being short or long. Resources are allocated in the same group if a request for one is nested in a request for the other. In other words if a task have nested critical sections, those requested resources will be located in the same group.

Each group is protected by a group lock. The group lock is a queue lock for the group of short resources and a semaphore for the group of long resources. In FMLP if a job wants to have access to a resource, it should request the resource's group lock. When the job requests a group lock it is queued in a FIFO queue to access the group lock. As soon as the job gets the lock and holds the resource, the request is satisfied.

Short requests: If a job requests for a short and outermost resource it first becomes

non-preemptable and then tries to get the queue lock of the resource. An outermost resource request is a critical section, which is not nested in any other critical section. If the lock is not available, the job performs a busy-wait while waiting in the FIFO queue for the resource. Once the task gets the resource lock, it can have access to the resource. After executing its critical section the job releases the group lock and leaves the non-preemptable section.

Long requests: If a job requests for a long and outermost resource it tries to get the semaphore lock of the resource group. If the lock is not available, it will be added to a FIFO queue and suspends itself. As soon as the job holds the resource group lock it resumes and become non-preemptable. The job's priority is boosted when it executes its critical section and when the execution is completed it releases the group lock and become preemptive again.

If more than one priority boosted job are ready, they will be served in a FIFO queue. In case of nested resources, either short or long, when a job issues a nested request, the request is satisfied immediately since the job already has the group lock.

1.2.5 OMLP Synchronization Protocol

Another locking protocol for handling resource sharing in multiprocessors is $O(m)$ Locking Protocol (OMLP) proposed by Brandenburg and Anderson [5]. OMLP has been developed for both partitioned and global scheduling.

OMLP is denoted as *suspension-oblivious* protocol. Under a suspension-oblivious protocol suspensions are assumed as execution which means suspended jobs are assumed to occupy processor. In contrast, suspension-based protocols are *suspension-aware* protocols in which true suspension of tasks is considered and suspended jobs are not assumed to execute when they are suspended.

Furthermore OMLP is implied as *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.

In global OMLP for each resource l_k there exist two queues. FQ_k is a FIFO-based queue with maximum length m , where m is the number of processors, and PQ_k is a priority-based queue in which tasks are enqueued when there are more than m jobs requesting the resource. The job in the head of the FQ_k holds the resource l_k . When a job sends a request for a resource if there are less than m jobs in the FQ_k it will be added to the FQ_k , otherwise it will be added to the PQ_k . All the jobs in both queues are suspended except the job in the head of the FQ_k which is ready and inherits the priority of the highest priority job located in FQ_k and PQ_k . When a job releases the resource it is dequeued from FQ_k and the next job in the head of FQ_k is resumed, meanwhile the highest priority

job in the PQ_k will be transferred to FQ_k .

According to the global OMLP, the lower priority jobs are prevented to be exhausted since they have a chance to be added to the FIFO-based queue, FQ_k . On the other hand, higher priority jobs will be punished for less than m lower priority jobs critical sections waiting in the priority-based queue.

Tasks compete on global resources under partitioned OMLP first have to acquire a unique token devoted to each processor. There is only one token in each processor used for all global resources requested by tasks in that processor. When a job requests a global resource if the token dedicated to the job's assigned processor is not held by any other job, then the job holds the token. Otherwise, it will be suspended and added to a priority-based local queue on that processor. As soon as the job gets the token it will be added to a global FIFO queue and its priority is boosted to the highest priority of any job in that processor. The job will be scheduled as soon as it is in the head of the FIFO queue. When the job releases the resource it is removed from the FIFO queue and releases the token, so the next job in the head of the FIFO queue (if any) is resumed. This limits the number of jobs that can cause priority inversion in the system since the job's priority is boosted when the job holds the token. Yet, the jobs that do not share resources may still incur priority inversion blocking due to priority boosting, which is not the case in global OMLP that utilizes the priority inheritance technique.

1.2.6 MSOS Synchronization Protocol

Multiprocessors Synchronization Protocol for real-time Open Systems (MSOS) is another synchronization mechanism for resource handling among independently-developed real-time applications presented by Nemati et al. [21]. Systems that are composed in this platform together are systems that 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 each global resource in a processor are handled by a local queue in which tasks requested the resource will be enqueued.

The local queues can be FIFO or priority based. As the global queue is a FIFO-based, whenever the resource is available for the processor in the head of the queue, the task in the head of its local queue will lock the resource. Depending on the local queue basis (being priority-based or FIFO-based) the task with the highest priority among the requesting tasks or the task which has requested sooner, will get access to the resource.

When a task on a processor requests a global resource, its priority is boosted immedi-

ately to its original priority plus the highest priority on that processor. In this way just the tasks with higher priorities which have got access to other resources, can preempt the task. Therefore, the waiting time of other tasks will be a function of only global critical sections.

When a task on a processor requests a global resource, if the resource is available, which means both global and local queues of the resource are empty, the task will lock the resource will be granted to the task. Otherwise a placeholder for the processor is placed in the global FIFO queue of the resource and the task will be enqueued in the local queue of the processor.

In the FIFO-based local queue a task will have access to the resource whenever its corresponding placeholder in the global queue is at the top of the queue, and the task itself is in the head of the local FIFO queue, which is not the case for the tasks in priority-based queue.

In the priority-based queue, a higher priority task which has requested a resource will have access to the resource earlier than a lower priority task that has already located a placeholder in the resource's global queue. In other words, the higher priority tasks will always have access to the global resources whenever the processor is in the head of the FIFO queue, despite the fact that any lower priority task has requested the resource earlier. This may cause long delay for the lower priority tasks.

Moreover, Easwaran and Andersson proposed a synchronization protocol [10] under the global fixed priority scheduling protocol. They have investigated schedulability analysis of the Priority Inheritance Protocol (PIP) under global scheduling algorithms.

The local resource synchronization protocol can be one of the uni-processor synchronization protocols such as PCP or SRP by T. Baker [2].

- PCP: Priority Ceiling Protocol (PCP) is an extension of Priority Inheritance Protocol, where each resource has a statically assigned priority ceiling value, defined as the maximum priority of all tasks that may request the resource. System ceiling on a processor is the maximum priority ceiling of all local resources currently used by other tasks on that processor. When a task requests a resource, the request is granted only if the priority of the task is greater than the system ceiling or if there are no other locked resources at the moment, also if the task holds the resource that raised the system ceiling last.
- SRP: Under stack resource policy (SRP) resource requests are served immediately. SRP is similar to the PCP with the difference that once a task starts executing it never gets blocked. A job may not execute after releasing its resource until its priority become greater than the system ceiling.

Both PCP and SRP protocols prevent deadlock and limit maximum length of priority inversion.

Inspiring from the described protocols, we have investigated two algorithms for handling resource requests under semi-partitioned scheduling regardless of the partitioning algorithms. In the rest of this report we focus on presenting the proposed algorithms and analyzing the relevant schedulability analysis.

2 Theoretical Background

It seems that ordinary scheduling algorithms are not effective enough to deal with complicated nature of multi-core architecture. In fact efficient algorithms are needed to consider timing constraints in the system. Two major conventional algorithms developed for scheduling real-time tasks on multiprocessors are categorized as global and partitioned scheduling, which we discuss them here in more detail. Later, we review the semi-partitioned approach and the latest algorithms for partitioning under this scheduling method.

2.1 Global Scheduling

In the global scheduling approach, a global ready queue is used to store all ready tasks in the system. The global scheduler then selects the highest priority tasks among all ready tasks in the ready queue for executing on the available processors.

The strength point of global scheduling approach is that jobs of a task are able to execute on different processors as well as different parts of a job which also can execute on different processors. However uniprocessor scheduling algorithms such as rate-monotonic (RM) and earliest deadline-first (EDF) [26] are not suitable for multiprocessor scheduling as they are not optimal on the multiprocessor platforms and alleviate the utilization.

In contrast, DP-FAIR scheduling guarantees the optimality on multiprocessor platforms [20]. However, the overhead introduced to the system by this method is considerably high compared to other approaches such as semi-partitioned scheduling. The scheduling scheme has been shown in Figure 1.

2.2 Partitioned Scheduling

In partitioned scheduling, each task is assigned to a single processor, on which all of its jobs will execute. For each processor a scheduler and a local ready queue is required to independently schedule the tasks on the processor. Different schedulers may utilize identical scheduling algorithm or use different ones.

Figure 1: Global scheduling

The main advantage of this approach is to reuse the well-known uniprocessor scheduling algorithms. One of the disadvantages of this approach is the fact that finding an optimal allocation algorithm is not possible in polynomial time. Assigning tasks to processors in multiprocessor system is a bin-packing problem which is a NP-hard problem. Therefore heuristics are used to partition tasks between processors such as first-fit strategy.

Global scheduling algorithms better utilize multiprocessor systems than partitioning approaches when overheads are negligible. On the other hand, global algorithms introduce large amount of overheads in large platforms[7]. The scheduling scheme has been shown in Figure 2.

Figure 2: Partitined scheduling

2.3 Semi-Partitioned Scheduling

Semi-partitioned scheduling is a mix between pure partitioned and global scheduling approaches which has been first introduced by Anderson et al. [1]. Semi-partitioned scheduling extends partitioned scheduling by allowing a low number of tasks to be split among different processors, thereby improving schedulability. Other tasks in the system are allocated to fixed processors same as partitioned scheduling. Similar to partitioned scheduling, semi-partitioned algorithm utilizes separate ready queues for each processor. The individual scheduler of each processor manages ready tasks from the ready queue to access the processor capacity.

Semi-partitioned scheduling is based on assigning most tasks in the system to each specific processor. These tasks are called split tasks. The tasks that cannot be completely assigned to one processor will be split up and allocated to different processors which are called non-split tasks. The process of assigning tasks to processors is done off-line. Different task allocation methods have been proposed in prior works [16, 17, 19, 14]. Guan et al. in [14] have increased the utilization of task sets to the utilization bound of RMS on single processor.

In this work Guan et al. introduced two partitioning algorithms for semi-partitioned scheduling. Both algorithms use RMS on each processor and assign tasks in decreasing period order. In contrast to previous assigning algorithms, both algorithms apply *worst-fit* partitioning while others use *first-fit*. In worst-fit, the algorithms select the processor with the least total utilization to assign the new tasks. Under these algorithms, the utilization of task sets are allowed to be as high as the utilization bound of Liu and Layland's RMS for any task set. In contrast to worst-fit, the first-fit algorithm selects a processor and assigns tasks to it until the capacity of the processor is filled. Then selects the next processor and repeat the procedure.

- **The first algorithm SPA1**

SPA1 is based on worst-fit partitioning, since the occupied capacitance of the processors are increased in this method. The tasks are selected to be assigned to processors from a priority-based queue in the increasing order. Under worst-fit partitioning, the processor with the minimal total utilization is always selected to dedicate a new task which would be the lowest priority task from the priority-based queue.

Under SPA1, when a task cannot be assigned entirely to a processor, it is split up in two subtasks such that the utilization of the current selected processor with utilization of the first subtask of the split task is equal to $\theta(N)$, where $\theta(N)$ is the utilization bound of RMS on single processor and is equal to $N(2^{\frac{1}{N}} - 1)$. N is the number of tasks on the processor.

The second subtask of the split task is inserted to the priority-based queue. As a

result a task is split only when the selected processor capacity is completely filled. Since the tasks are assigning in the increasing priority order, the split tasks generally will have higher priorities on their allocated processors under worst-fit partitioning. This is desirable as if split tasks get higher priorities on their assigned processors, they can have a better chance to be schedulable [14].

It is proved in [14], that all task sets containing light tasks are schedulable with SPA1 on M processors, if the total utilization of the task set is less or equal than θ_N . A light task set is a task set in which each task has a utilization no larger than $\theta(N)/(1 + \theta(N))$. This is because when a task has a very large utilization, if it is split under the assigning algorithm, its last subtask may get a low priority on its allocated processor, even with worst-fit partitioning. This situation can be seen in figure 3.

Figure 3: The last subtask of a split task with large utilization may have a low priority level

- **The second algorithm SPA2**

To solve the problem of tasks with high utilization, the second algorithm SPA2 has been presented. According to this algorithm, heavy tasks in the task set implied with utilization larger than $\theta(N)/(1 + \theta(N))$, will first be identified using a simple test.

Once they satisfied the test, they would be pre-assigned to the processors. Hence, the heavy tasks will not be split anymore, since they are first assigned to the processors, So the addressed problem is solved. However, tasks that do not satisfy the test are assigned (and probably split) later together with the light tasks.

The system assigning scheme has been shown in 4

Figure 4: Semi-partitioned scheduling

3 Thesis Goal

3.1 Aim of thesis

In semi-partitioned scheduling most tasks are assigned to one fixed processor while a few number of tasks are split among different processors.

As mentioned before, a semi-partitioned scheduling algorithm consists of three parts: the partitioning algorithm, which determines how to allocate tasks or split them if required, among processors, the scheduling algorithm, which determines how to schedule the tasks assigned to each processor, and synchronization protocol which manages tasks accessing to shared resources which is also the focus of this report.

One of the problems in multiprocessor systems is the blocking time caused because of different tasks competing each other for shared resources in the system. So in this case the need of a synchronization protocol for accessing the shared resources is magnified which is now not available for semi-partitioned scheduling protocol. Therefore in this thesis two algorithms are discussed for handling resource requests in semi-partitioned scheduling.

3.2 Assumptions

Some assumptions have been considered for both algorithms which will be described next:

- Nested critical sections are prohibited which means a job does not request for a shared resource while holding another. Since there is no nested requests, tasks in the system will not incur deadlock.
- The tasks that access global resources, execute their critical sections non-preemptively. This means that a task cannot be preempted by any other task under any condition during executing a global critical section (*gcs*).

4 System Model

4.1 Task Set Model

In this section we introduce the system model. The multiprocessor platform consists of a task set of n periodic tasks $\{\tau_1, \tau_2, \dots, \tau_n\}$ running on m processors $\{P_1, P_2, \dots, P_m\}$. 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., 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. The utilization of each task τ_i is defined as:

$$U_i = \frac{C_i}{T_i} \quad (1)$$

Further the utilization of a task set on each processor is defined as:

$$U_k = \sum_{\tau_i \in \tau(P_k)} \frac{C_i}{T_i} \quad (2)$$

where τ_{P_k} is the task set on processor P_k .

In terms of semi-partitioned approach, some tasks are assigned just to one processor which are called non-split tasks. On the other hand, some tasks which can not completely fit into one processor are split and allocated to more than one processor which 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 executing before the former subtask has completed.

In the example shown in Figure 5, 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 . Arrival time of the task τ_i is denoted by a and T_i shows T_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 (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 regarding 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

first proposed algorithm for a specific resource request in the split tasks, since it may happen in any subtask of the split task.

Figure 5: Subtasks in a split task

4.2 Resource Sharing

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 $R_{P_k}^L$ and $R_{P_k}^G$ respectively. Moreover, $C_{s_{i,q}}$ denotes the worst-case execution time of the longest critical section of task τ_i in which it requests the resource R_q . Furthermore, $n_{i,q}^G$ identifies the number of gcs of task τ_i , in which it requests the resource R_q .

5 Algorithm 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 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 [14].

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 fills the processor capacity in the sense that no more tasks can be allocated to the processor. The capacity criteria under which tasks are assigning 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 task could not fit completely in the next processor again, 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 processor, and other tasks still can be allocated to the processor, are 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 in the 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 requested 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. We have distinguished between two terms granting a resource and have or get access to the resource in this paper. We say a task has been 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, we say the task have or 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 request 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.

Generally, most computer tasks do not have a fixed execution time. taskss possible execution flows, 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, 11, 28]. 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 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 by definition are 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.

5.1 Resource queues structure

As mentioned in Section 5, a priority-based global queue is considered for each global resource in the system in which the tasks requesting 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. This local queue is also a 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 6.

Figure 6: Sytem 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. Granting access to a task happens when the resource is available at the moment that the task has requested it, or when the task has the highest priority among all tasks in the related global resource queue if any 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 there is no task waiting in the local resource queue and also no task is executing on the processor, the task which has been granted access to the resource immediately executes its critical section on the processor and releases the resource once it is completed.

In the Figure 6, R_i^G denotes the request of τ_i to R_q . As it can be seen, tasks requesting 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 tasks from different processors which have requested R_q . In this example, task τ_1 on processor P_1 and task τ_5 on processor P_2 which both have requested resource R_q 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 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. As it can be seen in Figure 6, 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 resource R_s and R_q has been granted to them but the processor was not yet free.

5.2 Migration-based Synchronization Protocol

Migration-based Resource Sharing Protocol under Semi-partitioned Scheduling (MRPS) 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 executes 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 source processor can have access to the processor capacity. Once the job executes its critical section at destination processor, it will migrate back to its dedicated processor (source processor). As a result of migrations, the subtasks will incur overhead mainly due to cache-related migration delay [3]. This overhead is caused due to additional cache misses that a job incurs when resuming execution after a migration.

As shown in Figure 7 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 dedicated processor P_2 . As it can be seen, two overheads are introduced during execution of τ_i^2 due to migration.

Figure 7: Resource handling in Migration-based Synchronization Protocol

Note that, successive subtasks of a split task have a constant offset regarding to the previous subtask's response time. Once each subtask of a split task is considered as an independent task, all subtasks are treated the same as non-split tasks. First the requesting task is enqueued in the related global prioritized queue and then when the resource was granted to the task, it is added to the prioritized local queue in the related 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 leading 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 accesses a global resource. Note that, Tasks are enqueued in the global and local resource queues with their original priority.

5.3 Non-Migration-based Synchronization Protocol

In the Non-Migration-based Resource Sharing Protocol under Semi-partitioned Scheduling (NMRPS), the resource requests by split tasks will be served on the processor that the request occurs and no migration happens. This implies that every time a job requests a resource it will be added to the resource queue by the processor in which the request happens.

Similar to the first algorithm each subtask of a split task is assumed to be an independent task having a constant offset caused by previous subtasks. In difference with the first algorithm all subtasks are assumed to access to global resources in their dedicated processors and thus incur delay to the local tasks due to accessing to global resources.

As it can be seen in Figure 8 subtask τ_i^2 executes its critical section on its dedicated processor P_2 in contrast with the previous solution depicted in the example shown in Figure 7. The process of handling the tasks requesting global resources over inserting them to global and local queues besides the queues structure is similar to the first algorithm. It should be considered that at any time only one subtask is assumed to be able to locate a request in a global queue.

Figure 8: Resource handling in Non-Migration-based Synchronization Protocol

Tasks in other processors may block a task requesting a global resource which is called remote blocking and those processors are called remote processors. In the case of split

tasks, subtasks are treated as individual normal tasks in the scheduling analysis.

According to the second algorithm, subtasks are tasks that share resources with other tasks in the system. Therefore, with regarding to the scheduling analysis the number of tasks which cause remote blocking to other tasks in the system are increased. 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 critical sections related to the subtasks of a split task.

6 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 be 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 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.

6.1 Local Blocking due to Local Resources

We denote n_i^G as the number of global critical sections *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 later block τ_i after it is resumed.

This kind of blocking can happen up to n_i^G times. 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 maximum $\left\lceil \frac{T_i}{T_j} \right\rceil$ jobs before τ_i is finished. In addition, each job also can 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 calculated as follows:

$$B_{i,1} = \min \left\{ n_i^G + 1, \sum_{\rho_j < \rho_i} \left\lceil T_i / T_j \right\rceil n_j^L(\tau_i) \right\} \max_{\substack{\rho_j < \rho_i \\ \wedge \tau_i, \tau_j \in P_k \\ \wedge R_l \in R_{P_k}^L \\ \wedge \rho_i \leq \text{ceil}(R_l)}} \{Cs_{j,l}\} \quad (3)$$

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

6.2 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 its non-*gcs* sections. 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 requested a global resource before τ_i arrives.

Similar to $B_{i,1}$ term, τ_j can release at most $\left\lceil \frac{T_i}{T_k} \right\rceil$ 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 term, denoted by $B_{i,2}$, is calculated as follows:

$$B_{i,2} = \sum_{\substack{\forall \rho_j < \rho_i \\ \wedge \tau_i, \tau_j \in P_k}} \left(\min \{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\} \max_{R_q \in R_{P_k}^G} \{Cs_{j,q}\} \right) \quad (4)$$

6.3 Remote Blocking

Whenever a task has to wait for global resources locked by the tasks on other processors, it incurs a remote blocking. In our proposed algorithms tasks may experience remote blocking from two groups of tasks:

6.3.1 Tasks with Lower Priority

It may happen that a task τ_i on processor P_k requests a global resource R_q which has already been locked by a lower priority task τ_j on processor P_r . In this situation τ_i has to wait until τ_j releases R_q . On the other hand, after τ_j locks R_q , it may wait in the local resource queue of P_r in which all tasks that have been granted access to other global resources are waiting. As soon as τ_j is the highest priority task in the local queue, it will access R_q . In worst-case situation τ_j has to wait for all higher priority tasks on 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:

$$B_{i,3} = \sum_{\substack{\forall R_q \in R_{P_k}^G \\ \wedge \tau_i \in \tau_{q,k}}} n_{i,q}^G RWRL_{q,k'} \quad (5)$$

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 τ_j that share global resources other than R_q . where $RWRL_{q,k'}$ is the waiting time for the resource q on remote processors of processor k due to tasks with lower priority than τ_i and is calculated as follows:

$$RWRL_{q,k'} = \max_{\substack{\forall \rho_j < \rho_i \\ \wedge \tau_j \in \tau_{q,r} \\ \wedge k \neq r}} \{Cs_{j,q} + wlh_{j,q}\} \quad (6)$$

$wlh_{j,q}$ is the waiting time introduced by τ_j 's local higher priority tasks on the τ_i 's remote processors. In other words, this is the time which τ_j has to wait for the processor because of higher priority tasks which have locked global resources other than R_q .

$$wlh_{j,q} = \rho_{h,j}(R'_q) \max_{\substack{\tau_t \in P_r \\ \wedge \rho_t > \rho_j \\ \wedge R_s \in R_r^G \\ \wedge s \neq q}} \{Cs_{t,s}\} \quad (7)$$

6.3.2 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 $\left\lceil \frac{T_i}{T_t} \right\rceil$ during the execution of τ_i 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 . τ_t may also wait in the local resource queue in P_r for at most one critical section per each higher priority task which has locked a global resource other than R_q . The related blocking term which is denoted as $B_{i,4}$ is calculated as follows:

$$B_{i,4} = \sum_{\substack{\forall R_q \in R_{P_k}^G \\ \wedge \tau_i \in \tau_{q,k}}} RWRH_{q,k',i} \quad (8)$$

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 .

$$RWRL_{q,k',i} = \sum_{\substack{\forall \rho_t > \rho_i \\ \wedge \tau_t \in \tau_{q,r} \\ \wedge r \neq k}} n_{t,q}^G \lceil T_i/T_t \rceil (Cs_{t,q} + wlh_{t,q}) \quad (9)$$

Total blocking of a task τ_i in the system is then calculated as below:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} \quad (10)$$

7 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 mostly due to cache-related overhead. The execution time of the migrated task is inflated by per migration overhead twice.

The reason is that the migrated task incurs a delay once when it migrates to the marked processor and once again when it has migrated back to its dedicated processor (7). The migrated critical section has to be inflated with one migration overhead as well. This is because, when a task is blocking by a migrated task, besides the critical section length, it also incurs an extra delay caused by the migration overhead. Therefore, this overhead is included in the critical section length.

$$C_{i_{migrated}} = C_i + C_{overhead} \quad (11)$$

$$Cs_{i_{migrated},q} = Cs_{i,q} + C_{overhead} \quad (12)$$

8 Remarks on blocking terms

8.1 Local Blocking due to Local Resources

In term $B_{i,1}$ in equation (3) which is a local blocking situation due to local resources, in both proposed algorithms the tasks which introduce local blocking, can not be of kind split tasks. This is because split tasks do not share local resources and as mentioned before all their resources are global. However, a split task may incur this type of blocking from the tasks requesting local resources with a ceiling higher than the priority of the split task. According to different scenarios under each of the two algorithms, we discuss here the blocking situations for split tasks in more detail.

- **Under MRPS:** In this algorithm when a subtask in a marked processor is blocked on a global resource, similar to non-split tasks it suspends and gives the chance to lower priority tasks to lock local resources which in turn may block the subtask in its non-critical sections. The number of global critical sections of the subtask is assumed to be the same as the original split task, since in worst-case all critical sections may occur in any of the subtasks.

For the subtasks which will migrate to marked processor, the number of global critical sections is assumed to be the same as the original task. This is regarding to the fact that this blocking term occurs due to suspension on the resource and it makes no difference that the task executes its gcs's in another processor rather than

its own assigned processor. Note that, the lower priority tasks which cause blocking because of locking local resources are on the subtask's assigned processor, not the processor it has migrated to.

- **Under NMRPS:** Under the second algorithm since all subtasks of a split task execute their critical sections in their own processor, the blocking situation is identical for all subtasks. If the subtask of a split task is suspended on a global resource, lower priority tasks get the chance to lock local resources. As all critical sections requested by the original split task may happen in any of its subtasks, therefore the number of global critical sections for each subtask is assumed equal to the number of critical sections of the original task.

8.2 Local Blocking due to Global Resources

In term $B_{i,2}$ in equation (4) which is local blocking due to global resources, in both proposed algorithms the tasks which introduce local blocking and those that get blocked can be split tasks. Depending on different scenarios under each of the two algorithms, we discuss here the blocking situations for split tasks in more detail.

- **Under MRPS:** In this algorithm when a subtask in the marked processor is blocked on a global resource, similar to non-split tasks, it gives the chance to lower priority tasks to lock other global resources which in turn may block the subtask in its non-critical sections. The number of global critical sections of the subtask is assumed to be the same as the original task, since all the critical sections may occur in any of the subtasks.

Similar to the first blocking situation, for subtasks which will migrate to the marked processor, the number of global critical sections is assumed to be the same as the original task regarding to the reason mentioned before. The lower priority tasks which cause blocking because of locking other global resources are on the subtask's assigned processor, not the processor it has migrated to.

- **Under NMRPS:**

Under the second algorithm since all subtasks of a split task execute their critical sections in their own processors, the blocking situation is identical for all subtasks. If the subtask of a split task is suspended on a global resource, the lower priority tasks get the chance to lock other global resources on the subtask's processor. As all critical sections requested by the original task may happen in any of its subtasks, as the number of global critical sections are assumed to be equal to the number of gcscs of the original task.

8.3 Remote Blocking

In terms $B_{i,3}$ and $B_{i,4}$, in Equations 5 and 8 respectively, (which are the blocking incurred from the lower and higher priority tasks on remote processors) in both algorithms the tasks introduce remote blocking and those that get blocked remotely, can be split tasks. According to different scenarios under each of the two algorithms, we discuss here the blocking situations for split tasks in more detail.

- **Under MRPS:** In MRPS algorithm, a subtask in a marked processor remotely blocked on a global resource experiences the same situation as non-split tasks. However other subtasks, i.e., those that are not on the marked processor is of kind other subtasks rather than the subtask of marked processor, the remote processors for all the subtasks are the, since the task requesting a global resource migrates to the marked processor. In other words, the remote blocking time introduced to all the subtasks is the same.

In MRPS, any critical section of any subtask will execute on one processor, i.e., marked processor. Besides, any critical section occurs in only one subtask. Thus, from the remote tasks point of view, the total number of critical sections equals to the number of critical sections of the original task. In other words the remote tasks only need to consider one task with which they share resources.

- **Under NMRPS:** However, in non-migration-based protocol as all subtasks will execute the critical sections on their processors, the number of remote tasks introducing remote blocking in the system increases compared to the migration-based method. Since all critical sections of a split task may occur in any of its subtasks, we should assume that all critical sections of the original task are happening in all processors containing the subtasks.

Therefore, from the remote tasks point of view, the total number of critical sections equals to the number of critical sections of the original task multiplied by the number of subtasks. In other words the remote tasks have to consider all subtasks since the critical sections may happen in any of the subtasks. This is the reason which cause greater remote blocking under NMRPS algorithm.

9 Response Time Analysis

Response time analysis calculates the response times of the tasks and compares it to their deadlines, which in our case is equal to the period of the tasks since in the task model, tasks have implicit deadlines. A task τ_i is schedulable if:

$$R_i \leq T_i \quad (13)$$

R_i is the worst-case response time of task τ_i , i.e., the longest response time of all task instances. Hence, if the task instance with the worst-case response time will finish before its period, then all other instances of the same task will also do. A task set is schedulable if response times of all tasks in the set are less than or equal to their assigned periods.

When calculating the response time of a task τ_i the worst case will occur when all higher priority tasks releases at the same time. The response time of a task τ_i in its assigned processor is calculated as below:

$$R_i^{n+1} = C_i + B_i + I_i \quad (14)$$

The term I_i is called interference which is the pre-emption from higher priority tasks on the same processor. B_i is the task's total blocking time which has already been calculated in equation 10. I_i is calculated as below for the non-split tasks and the subtasks on first processors in the system:

$$I_i = \sum_{\rho_j > \rho_i} \lceil \frac{R^n}{T_j} \rceil C_i \quad (15)$$

Howevre, I_i is calculated as in equation 16 for other subtasks on a processor different from the marked processor, as they have a constant offset. The offset of the subtask O_i is equal to the maximum response time of the previous subtask of the split task.

$$I_i = \sum_{\rho_j > \rho_i} \lceil \frac{R^n + O_i}{T_j} \rceil C_i \quad (16)$$

The iteration is stopped when $R_i^{n+1} = R_i^n$ or $R_i^{n+1} > T_i$, and if $R_i \leq T_i$, we conclude that the task is schedulable.

10 Evaluation

In this section we present our experimental evaluation for comparison of the performance of MRPS and NMRPS. 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 the same and is based on first-fit partitioning.

The partitioning algorithm in a platform selects a processor and fill 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 to be assigned to the processor does not fit in completely, it is split in two subtasks. The first subtask fills the current processor capacity and next subtask will be assigned to the next processor under same procedure. The platform partitioning scheme can be seen in Figure 9.

Figure 9: Assigning mechanism based on first-fit partitioning

In order to be able to compare both algorithms based on resource sharing mechanism in terms of schedulability under identical conditions, we first generate platform for MRPS and evaluate the schedulability of the platform. Afterwards, the platform is changed upon NMRPS and again the schedulability is checked. So that in this way the resource allocation to different tasks in the platform is the same in both algorithms and the comparison is validated.

The experimental results show that the migration overheads in MRPS has a greater effect compared to the higher blocking terms of NMRPS in making the platforms unschedulable. With big overhead values, generally the platforms become more unschedulable under MRPS. This is an important factor which makes the MRPS algorithm suitable for multiprocessor platforms with tasks having small working set size (*WSS*) [3].

10.1 Experimental setup

As mentioned before in our experiments, we have employed same resource allocating and set ups to determine the schedulability performance under both algorithms. We randomly generated 1000'000 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. The number of resources which are shared among tasks in a platform is constant in each generated platform and is 10. 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 selected randomly via the range 5, 25, 45, 65, 85, 105, 125, 145, 165, 185, 205.

Setting Parameter	Defined range
Number of Processors per Platform	4, 8, 12, 16
Processor Maximum Utilization Capacity	0.3, 0.4, 0.5, 0.6
Number of Critical Sections per Task	1, 2, 3, 4, 5, 6
Critical Sections Length	5, 25, 45, 65, 85, 105, 125, 145, 165, 185, 205
Per Migration Overhead in MRPS	0, 20, 60, 140, 300, 620, 1260, 2540
Number of Resources per Platform	10
Minimum Task Period	10
Maximum Task Period	100
Minimum Task Utilization	0.01
Maximum Task Utilization	0.101

Table 1: Platform parameters setting range

The migration overhead which is mostly the result of cache-related delay has been chosen randomly for MRPS algorithm from a set of values 0, 20, 60, 140, 300, 620, 1260, 2540 providing small overheads to large amounts. The small overhead values has been included in the set up in order to compare the MRPS algorithm behavior with negligible overhead amounts over NMRPS performance. On the other hand, with the big values we study the effect of large overhead on schedulability performance under MRPS. The set of values specified for resource sharing parameters in generated platforms has been shown in Table 1.

The experiments have been done for 1000'000 platforms and repeated more than 3 times which each time yielded mostly in same results. This shows that this 1000'000 samples can be representative.

10.2 Results

In this section we present the evaluation results of our experiments for both algorithms. The Figures depict schedulability 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, and the value of per-migration overhead. The results show that the migration overhead play a prominent role in degrading the performance of MRPS. On the other hand ignoring the overhead, MRPS performs better.

The results depicted in Figure 10 show that by increasing the critical section lengths the schedulability decreases. Increasing the critical sections will increase the blocking durations of other tasks on resources and may cause some tasks in the platform miss their deadlines. However, considering the migration overhead the schedulability rate decreases

in MRPS algorithm compared to NMRPS. As one expect, the behavior of both algorithms versus changes in critical section is almost the same.

Furthermore, the number of critical sections per task also affects the schedulability. As it can be seen in Figure 12, as the number of critical sections per task increases the platforms become more unschedulable. This is an expected result as increasing the resource requests in tasks the blocking on resources also increase and as a result, affects the schedulability negatively. The schedulability under MRPS against the migration overhead has been presented in 11.

Number of processors is also a determinant factor for schedulability in a multiprocessor platform. As it can be seen in Figure 13, as the number of processors increases the schedulability percentage decreases. This is also expected since the number of resources is constant in the platforms, and more processors means more tasks competing over a constant number of resources. Under this circumstances the blocking on resources increases which will conclude in less schedulability. Although the behavior of both algorithms in terms of changes in the number of processors in the platform is overallly identical, yet MRPS perform better without considering the migration overhead. By increasing the overhead NMRPS performs better for higher numbers of processors, e.g., above 8 processors.

Shown in Figure 14, the capacity of the processors in a platform also affects the system performance. As the utilization capacity of the processors increases the schedulability decreases. This was an expected result because when the processor capacities decreases, fewer tasks can be assigned to them. This means that less tasks will compete over resources in the systems which will result in less blocking on resources and finally increasing the schedulability.

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

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

(a) Overhead = 0 μ s

(b) Overhead = 140 μ s

Figure 10: 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

Figure 11: 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

11 Conclusions and Future Works

11.1 Summary and Conclusion

In this thesis, we proposed two synchronization protocols under semi-partitioned scheduling for multiprocessor platforms. The protocols handle the resource sharing among tasks while some tasks have been allocated to more than one processor.

In the first proposed algorithm, the critical sections of a split task are executed on one of the processors which the related subtask can fit all the critical sections. However, in the second algorithm, the critical sections will be executed on the same processor on which the subtask requests the resource.

According to the global and local queues structure for handling the global resources, the blocking terms under both algorithms have been studied. The analysis conducted to a greater remote blocking under the second algorithm since the number of tasks causing remote blocking in the system decreases in the first algorithm due to centralizing requests on one processor for split tasks.

The experimental evaluation have been concluded to study the performance of the proposed algorithms. The results confirm the better performance of the first algorithm in handling the resource requests under low migration overheads. However, the first algo-

(a) Overhead = 0 μ s

(b) Overhead = 20 μ s

(c) Overhead = 140 μ s

Figure 12: 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

(a) Overhead = 0 μ s

(b) Overhead = 60 μ s

Figure 13: 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

(a) Overhead = 0 μ s

(b) Overhead = 60 μ s

Figure 14: 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

rithm's performance degrades under larger overheads.

This leads to the conclusion that MRPS suits better for multiprocessor platforms with lower migration overhead. While NMRPS performs better in platforms with high migration overhead.

However, there are some determinant factors that should be considered while generating a platform. These factors which has been identified through the experimental results are, the number of processors in the platform, the number of resources requested by each task in the system, the length of critical section in which the task use a resource, the utilization capacity dedicated to each processor and the migration overhead.

11.2 Future works

The work presented in this thesis can be extended in different aspects:

1. Propose a proper allocation mechanism in the presence of resource sharing similar to techniques proposed in [18] [22] in the system to deliver a complete solution. The partitioning algorithm can not be an optimum solution, but a good heuristic can be devised to not only handles the resource requests in the system well, but also improve the utilization bounds.
2. Implement the solutions in a real operating system and compare the results with theoretical evaluation.
3. Extend the solution to support task sets of explicit deadlines.
4. Design the queue structures of handling resources in a way that bounds the blocking durations to a constant factor of blocking similar to OMLP.
5. Propose other solutions for handling resource sharing under semi-partitioned scheduling to improve the performance.

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*, pages 199 – 208, july 2005.
- [2] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.
- [3] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 125 –135, july 2011.
- [4] 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*, pages 47 –56, aug. 2007.
- [5] B. B. Br and J. H. Anderson. Optimality results for multiprocessor real-time locking.
- [6] 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.
- [7] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 247 –258, july 2007.
- [8] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS*. Chapman Hall/CRC, Boca, 2004.
- [9] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/~anderson/diss/devidiss.pdf*, 2006.
- [10] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.
- [11] 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.

- [12] 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*, pages 189 – 198, may 2003.
- [13] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *proceedings of 22nd IEEE Real-Time Systems Symposium*, pages 73–83, 2001.
- [14] 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*, pages 165 –174, april 2010.
- [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.
- [16] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, april 2008.
- [17] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 23 ,32, april 2009.
- [18] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *proceedings of 30th IEEE Real-Time Systems Symposium (RTSS’09)*, pages 469–478, 2009.
- [19] 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*, pages 239 –248, july 2009.
- [20] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 3 –13, july 2010.
- [21] 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*, pages 251 –261, july 2011.
- [22] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, pages 253–269, 2010.

- [23] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116 –123, may-1 jun 1990.
- [24] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
- [25] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259 –269, dec 1988.
- [26] C. S.Dhall. On a real-time scheduling problem, 1978.
- [27] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175 –1185, sep 1990.
- [28] 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.*, 7(3):36:1–36:53, May 2008.