Semi-Partitioning under a Blocking-Aware Task Allocation

Sara Afshar, Moris Behnam, Thomas Nolte
Mälardalen University, Västerås, Sweden
Email: {sara.afshar, moris.behnam, thomas.nolte}@mdh.se

***Abstract*—Semi-partitioned scheduling is a resource efficient scheduling approach compared to the conventional multiprocessor scheduling approaches in terms of system utilization and migration overhead. Semi-partitioned scheduling can better utilize processor bandwidth compared to the partitioned scheduling while introducing less overhead compared to the global scheduling. Various techniques have been proposed to schedule tasks in a semi-partitioned environment, however, they have used blocking-agnostic allocation mechanisms in presence of resource sharing protocols. Since, the allocation mechanism can highly affect the system schedulability, in this paper we provide a blocking-aware allocation mechanism for semi-partitioned scheduling framework under a suspension-based resource sharing protocol. We have applied new heuristics for sorting the tasks in the algorithm that shows improvements upon system schedulability. Finally, we present our preliminary results.**

I. Introduction

The interest of migrating towards multi core platforms as the defacto processors in the industry has raised an inevitable demand to investigate different aspects of multiprocessors in terms of system performance. One concern within this context is constraints on available resources in the system. Embedded systems with limited resources are more subject to performance degradation. For such systems, resource optimization approaches play an important role. Semi-partitioned scheduling approach is a resource efficient scheduling approach compared to the conventional multiprocessor scheduling approaches, global and partitioned scheduling.

Under semi-partitioned scheduling which was first introduced by Anderson et al. [3], during allocation phase tasks are assigned to processors similar to partitioned approach. When partitioned scheduling fails to assign tasks, semi-partitioned approach will continue task allocation by splitting the task that cannot completely fit into one processor and allocate it to different processors. In this way the processors can be utilized more efficiently.

Despite the fact that semi-partitioned scheduling benefits from higher system utilization by allowing some tasks to migrate among processors, yet it deals with task partitioning problem where finding an optimal solution is known to be NP-hard. The partitioning of tasks in the system can highly affect the system performance. In all previous works done in semi-partitioned context both without considering resource sharing [7], [8], [10], [6] and in presence of resource sharing [2], [1], conventional task allocation mechanisms has been used to assign tasks to the processors such as first-fit or worst-fit. To the best of our knowledge no work has studied advanced allocation mechanisms for semi-partitioned scheduling where the allocation is performed based on the guaranteed system schedulability. In this work, we propose a new framework for semi-partitioned approach that uses a smart allocation mechanism which takes the effect of resource sharing into account while keeping the system schedulable.

Previously, several blocking-aware task allocation techniques have been proposed for partitioned scheduled systems. Lakshmanan et al. have proposed a blocking-aware partitioning algorithm called SPA [9]. Nemati et al. have proposed a blocking-aware partitioning algorithm called BPA [12] which showed higher performance compared to SPA. For our proposed scheduling framework we have chosen one recent partitioning heuristic proposed by Wieder and Brandenburg [13] which is called Greedy Slacker and has shown better results compared to the existing methods while providing more transparency for the allocation algorithm. We leave the comparison with other allocation techniques to find the most proper allocation approach for semi-partitioning as future work.

The key idea behind greedy slacker algorithm is to assign a task to the processor with the maximum provided slack during allocation phase. In order for the algorithm to work more efficient, during each assignment of a task to a processor the priorities of tasks on that processor are revised based on OPA *(optimal priority assignment)* [4]. The algorithm has been applied under a spin-based resource sharing protocol. In this paper, we have extended the semi-partitioned scheduling under a suspension-based protocol using a blocking-aware allocation technique. We have proposed two new sorting mechanism for the algorithm other than the one presented in [13] to sort tasks prior to allocation which showed improvements in system schedulability. Finally, we present our preliminary experimental results.

II. System Model

Our system consists of m identical processors with n sporadic tasks. Each task τ_i is identified by a worst-case execution time C_i, the minimum inter arrival time T_i and a deadline D_i, in which $C_i \leq D_i \leq T_i$. The priority of a task τ_i is denoted by ρ_i. In our model a task τ_i has a higher priority than that of task τ_j, if $i > j$ ($\rho_i > \rho_j$).

Based on the semi-partitioned approach some tasks in the system are assigned just to one processor that are referred as non-split tasks. The tasks which cannot completely fit into one processor are split during the partitioning phase and may allocate on more than one processor depending on how many processors they need. These type of tasks are denoted as split tasks where each single part of a split task is called subtask. For the sake of simplicity we refer to non-split tasks as tasks in the rest of this paper. All subtasks of a split task are assumed as a normal task in the system. However, since tasks execution in our system model conforms to a serial execution model, therefore each subtask of a split task should finish its execution prior to its successive subtasks. A split task τ_i is modeled by l subtasks $(\tau_i^1,...,\tau_i^l)$ where each subtask is represented by $(C_i^k, T_i, O_i^k, \rho_i^k)$ and $k = 1,...,l$. $\rho_{P_k}^{\max}$ denotes the maximum priority level among tasks (or subtasks of split tasks) on a processor P_k.

O_i^k denotes a constant offset for the k^{th} subtask of a split task τ_i which represents the delay that each subtask should take into account before start executing. Assigning the maximum response time of the former subtask's as the offset guarantees each subtask start executing after all previous subtasks are finished. The first subtask of a split task does not need to wait therefore $O_i^1 = 0$. The group of tasks (including subtasks of split tasks) assigned to a processor P_k are represented by $\mathcal{T}_{P_k}$.

Tasks may use *local* or *global* resources. Resources that are accessed only by tasks on the same processor are denoted as local resources and resources that are requested by tasks on different processors are denoted as global resources. Based on this definition all resources requested by split tasks are assumed global resources since a request of a split task may happen on any of its assigned processors due to the fact that different instances of the same task may have different execution lengths. Local and global resource requests by jobs of task τ_i are denoted by $\mathcal{RS}_i^{\mathrm{L}}$ and $\mathcal{RS}_i^{\mathrm{G}}$, respectively. Further, we denote $Cs_{i,q}$ as the worst-case execution time among all requests of any job of a task τ_i on a resource R_q. We also define n_i^{G} and $n_{i,q}^{\mathrm{G}}$ as the maximum number of requests for any global resource and a specific global resource R_q by any job of task τ_i, respectively. Nested access of resources is not handled in the system model, however, it can be adjusted using group locks similar to [5].

III. Resource Sharing

We use the suspension-based resource sharing protocol as in [11]. Next we briefly recapitulate the rules of this protocol. Rule 5 has been provided due to semi-partitioned approach and is similar to Rule 12 in [1].

Rule 1: Uniprocessor synchronization protocol, such as PCP or SRP are used for handling request to local resources.

Rule 2: For each global resource one unique global queue is dedicated. The task that is blocked on a resource, i.e., its request is not satisfied since the resource is not available, is suspended and its request is placed in the related resource queue. The queue policy for handling resource requests is FIFO.

Rule 3: When a global resource is released the task which its request is at the head of the related resource queue resumes and locks the resource and its priority is boosted to hasten the release of the resource. The task retrieves its original priority as soon as it finishes its global critical section *gcs* (the part of task execution that uses a global resource).

Rule 4: A task τ_i' priority is boosted to the highest normal priority on its assigned processor P_k while keeping its relation to its original priority i.e. $\rho_{P_k}^{\max} + \rho_i$. The key idea is that a task that accesses a global resource can preempt any task executing a non-gcs part as well as tasks with lower priority executing a gcs.

Rule 5: Migration to the next subtask of a split task will be postponed if the subtask is holding a resource. The subtask migrates as soon as the resource is released. The maximum amount for such execution should be considered either by inflating the execution time of the task with such amount or including in the processor slack (see SectionIV-B) .

Rule 6: Whenever a task completes its gcs, it releases the resource and the resource becomes available for the next task in the resource queue if any.

IV. Blocking Analysis Recap

In this section we briefly recapitulate the blocking terms that a task may experience under the suspension-based protocol presented in [11] which is used here.

Whenever a task sends a request for a global resource and the resource is not available the task gets blocked on the resource. There are different situations in which a task may experience blocking which we present briefly as below.

A. Local Blocking

Whenever a task τ_i is suspended due to requesting a global resource which is held by another task (on a remote processor) or before τ_i arrives, a lower priority task can request a local resource. A local resource sharing protocol such as PCP assigns a ceiling to each local resource $ceil(R_\ell) = \max\{\forall \rho_i | \tau_i \in \mathcal{T}_{P_k} \wedge R_\ell \in \mathcal{RS}_i^{\mathrm{L}}\}$. If the ceiling of the requested local resource is higher than the priority of τ_i then the lower priority task can contribute in delaying τ_i in τ_i's non-gcs part. This type of blocking is referred to as *local blocking due to local resources* which we denote it as B^{LL} and is upper bounded as follows.

$$B_i^{LL} = \min\{n_i^G + 1, \sum_{\substack{\rho_j < \rho_i \\ \wedge\, \tau_i, \tau_j \in \mathcal{T}_{P_k}}} \Big((\lceil \frac{T_i}{T_j} \rceil + 1) n_j^{\mathrm{L}}(\tau_i) \Big) \} \times \max_{\substack{R_l \in \mathcal{RS}_j^{\mathrm{L}} \\ \wedge\, \rho_i \le ceil(R_l)}} \{Cs_{j,l}\}. \quad (1)$$

where $n_j^{\mathrm{L}}(\tau_i)$ denotes the number of the critical sections in which τ_j requests local resources with ceiling higher than ρ_i.

Similar to the scenario described above, whenever a task τ_i is blocked on a global resource (and is suspended) or before its arrival, a task with lower priority can get the chance to request a global resource. According to Rule 4, the priority of a task granted access to a global resource is boosted to higher than any normal priority on the processor. As a result, τ_i can be delayed by gcs of such task with a lower priority when τ_i finishes its gcs and enters a non-gcs part. We refer to this type of blocking as *local blocking due to global resources* which is denoted by B^{LG} and is upper bounded as follows.

$$B_i^{LG} = \sum_{\substack{\forall \rho_j < \rho_i \\ \wedge \tau_i, \tau_j \in \mathcal{T}_{P_k}}} \Big(\min\{n_i^G + 1, (\lceil \frac{T_i}{T_j} \rceil + 1) n_j^{\mathrm{G}}(\tau_i)\} \times \max_{R_q \in \mathcal{RS}_j^{\mathrm{G}}} \{Cs_{j,q}\} \Big). \quad (2)$$

1) Remote Blocking: The request of a task for a global resource may not be satisfied since another task on a remote processor has already locked the same resource which is referred to as *remote blocking* and for a task τ_i is denoted by B_i^R. The maximum remote blocking delay is incurred to a task τ_i under the worst case scenario where all tasks on all remote processors that share resource with τ_i, request the resource earlier than τ_i. The maximum time that a resource R_q can be locked by a processor P_r is denoted by $RLT_{q,r}$ and is the accumulative time that all tasks on P_r can hold R_q . Each task τ_j on P_r that uses a resource R_q can in the worst case lock the resource for its maximum critical section on R_q and the maximum interference from higher priority tasks that are granted other global resources which is presented as follows.

$$RLT_{q,r} = \sum_{\substack{\forall \tau_j \in \mathcal{T}_{P_r} \\ \wedge R_q \in \mathcal{RS}_j^G}} \big(Cs_{j,q} + \sum_{\substack{\forall \rho_h > \rho_j \wedge \tau_h \in \mathcal{T}_{P_r} \\ \wedge R_s \in \mathcal{RS}_h^G \wedge q \neq s}} Cs_{h,s} \big). \quad (3)$$

Based on this, the maximum remote blocking that a task τ_i on a processor P_k may experience is upper bounded as follows:

$$B_i^R = \sum_{\substack{\forall R_q \in R_{P_k}^G \\ \wedge \tau_i \in \mathcal{T}_{P_k}}} \big(n_{i,q}^G \times \sum_{\forall P_r \neq P_k} RLT_{q,r} \big). \quad (4)$$

The total blocking that a task τ_i may experience is calculated as follows:

$$B_i = B_i^{LL} + B_i^{LG} + B_i^R. \quad (5)$$

B. Smart Allocation mechanism

In this section we present our algorithm which uses two rounds for task allocation. In the first round, the Greedy Slacker algorithm presented in [13] is used to allocate tasks until no task can be allocated based on a partitioned scheduling approach. In the second round, tasks that could not be allocated in the first round are allocated based on a semi-partitioning approach. Next, we briefly explain the Greedy Slacker algorithm first which is used in the first round. In the first round of the algorithm tasks are sorted based on decreasing order of their density (Line 3 in Alg. 1). We first define the notion of task slack and processor slack that we use in this paper. We denote slack of a task as the difference between its worst-case response time and its relative deadline. For the processor slack we use the minimum task slack among the tasks assigned to that processor. For every task from the ordered set, a processor is selected from the list of the processors in the system to which the task is assigned. The candidate processor is the one such that after allocation of the task to all processors will provide the maximal slack (Line 12 in Alg. 1). To check if a task τ_i can be assigned to a processor P_k, the tryAssign(τ_i, P_k) function of which the detail explanation can be found in [13] is used (Line 5 in Alg. 1). tryAssign(τ_i, P_k) by using OPA [4] reassigns priorities to all tasks of P_k when it assigns a new task τ_i to P_k, starting from the lowest priority to the highest priority level. For each assigned priority level it is checked if all tasks on the allocated processor and on remote processors are schedulable. Otherwise, tasks are removed from that priority level and are checked for the next priority level. The previous algorithm returned failure and stopped if no processor with remaining slack is found to assign a task whereas our algorithm will move to the second round.

When a task cannot be assigned to any processor in the first round it is inserted to a queue called Q_{splt} (Line 10 in Alg. 1) and is tried to be assigned to processors by splitting the task in the second round. After all tasks are allocated in the first round of the algorithm the remaining tasks are tried to be assigned to the processors which have remaining slack (Lines 18 to 23 in Alg. 1). findSlack(P_k) in line 19 returns the processor slack of P_k. The first processor that is selected to assign a candidate task for splitting is the one that has the maximum remaining slack (Line 27 in Alg.1). The execution time of the candidate task splits such that the first part uses the whole remaining slack of the selected processor except the amount equal to the maximum critical section of the task (Line 29 in Alg. 1). The reason for deducting the maximum critical section from the slack to assign as task's execution time is that according to Rule 5 a split task may need to overrun just before migrating to its next subtask. Therefore, a maximum amount for such execution is predetermined from the processor slack. In the second round, if no processor can be found that can accommodate a subtask of a split task the algorithm returns failure and stops (Line 25 in Alg. 1). For the sorting heuristic, besides the one based on density, we have tried two other sorting based on tasks' execution time and execution time plus remote blocking incurred to the task. In the next section, we show under the examined experiment set up that the two new sorting heuristics have better schedulability results compared to the one based on density.

V. Evaluation

In this section we demonstrate our preliminary experimental results. In our experiments we have evaluated the schedulability performance of the system for the proposed algorithm (Alg. 1) based on three sorting heuristics which are based on sorting tasks according to: *(i)* density, *(ii)* remote blocking and *(iii)* remote blocking and execution time which are presented in the graphs by *Dens*, *Blk* and *BlkExc*, respectively. In the experiments, we randomly generate a set of multiprocessor systems and task sets. The systems are evaluated based on the suspension-based algorithm presented in Sections III and IV.

A. Experimental Setup

In each experiment we have randomly generated 100 platforms. The number of processors are selected form a set {4, 8, 12, 16}. The number of used resources per platform is fixed to 10, and the number of resources used per task is selected from a set {2, 3, 4, 5}. Further, the length of each critical section is randomly generated from the range [1, 100] μS. We have generated each task set with per task utilization randomly generated from the range [0.1, 1] and the per task inter arrival time from the range [20,1000000] μS.

B. Results

We present our preliminary schedulability results based on selection of task set cardinality n and the number of the critical sections used per task *CsNum* under three applied sorting heuristics. The results has been illustrated in Figures 1 and 2.

Algorithm 1 Blocking-aware semi-partitioning heuristic

```
 1: Initialize $\mathcal{C} \leftarrow \emptyset$
 2: Initialize $\mathcal{D} \leftarrow \emptyset$
 3: for all $\tau_i \in$ order of decreasing density do
 4:     for all $P_k$ do
 5:         $s \leftarrow$ tryAssign($\tau_i, P_k$)
 6:         if $slack \geq 0$ then
 7:             $\mathcal{C} \leftarrow \mathcal{C} \cup (P_k, s)$
 8:         end if
 9:         if $|\mathcal{C}| == \emptyset$ then
10:             Add $\tau_i$ to $Q_{spilt}$
11:         else
12:             choose $(P_k, s)$ from $\mathcal{C}$ such that $s$ is maximal
13:             assign $\tau_i$ to $P_k$
14:         end if
15:     end for
16: end for
17: for all $\tau_j$ in $Q_{splt}$ do
18:     for all $P_k$ do
19:         $s \leftarrow$ findSlack($P_k$)
20:         if $slack > 0$ then
21:             $\mathcal{D} \leftarrow \mathcal{D} \cup (P_k, s)$
22:         end if
23:     end for
24:     if $|\mathcal{D}| == \emptyset$ then
25:         return Failure
26:     else
27:         choose $(P_k, s)$ from $\mathcal{D}$ such that $s$ is maximal
28:         Split $\tau_j$ into two parts $\tau_j^k$ and $\tau_j^{k+1}$
29:         $C_j^k \leftarrow s - maxCs_j$
30:         $C_j^{k+1} \leftarrow C_i - C_j^k$
31:         assign $\tau_j^k$ to $P_k$
32:         PushFront($\tau_j^{k+1}, Q_{splt}$)
33:     end if
34: end for
35: return true
```

The experiments show higher schedulability ratio under the two sorting heuristics based on remote blocking, and remote blocking plus execution time compared to the sorting based on density.

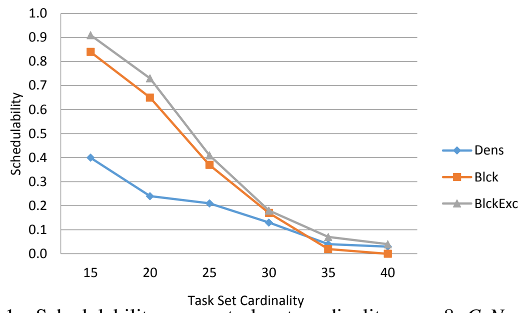

Fig. 1. Schedulability versus task set cardinality, $m = 8$, $CsNum = 2$.

VI. Conclusions and Future Works

In this paper, we proposed a blocking aware task allocation technique under a semi-partitioned approach. Our proposed algorithm uses two rounds for task allocation. In the first round we use a recently proposed blocking-aware approach that has been introduced for partitioned scheduled systems and has been applied under a spin-based resource sharing protocol. In this paper, we have applied the algorithm under a suspension-based resource sharing approach. The algorithm continues in the second round using a semi-partitioning approach splitting

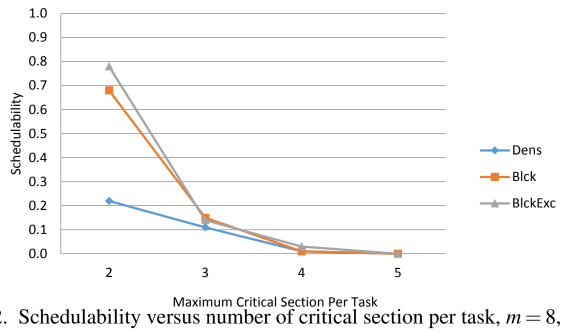

Fig. 2. Schedulability versus number of critical section per task, $m = 8$, $n = 20$.

the tasks which could not be allocated in the first round. We have proposed two new heuristics for sorting tasks which have shown an increase in the system schedulability under our preliminary experimental results. As future work we plan to perform extensive evaluations to further elaborate the system behavior. We also plan to compare our results with the existing solutions by adjusting them for semi-partitioned scheduling.

References

[1] S. Afshar, F. Nemati, and T. Nolte. Resource sharing under multiprocessor semi-partitioned scheduling. In 18th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 290–299, Aug. 2012.

[2] S. Afshar, F. Nemati, and T. Nolte. Towards resource sharing under multiprocessor semi-partitioned scheduling. In 7th *IEEE International Symposium on Industrial Embedded Systems (SIES), Work-in-Progress (WiP) session*, Jun. 2012.

[3] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In 17th *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 199–208, Jul. 2005.

[4] N. Audsley and Y. Dd. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

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

[6] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In 16th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 165–174, Apr. 2010.

[7] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*, pages 1–12, Apr. 2008.

[8] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In 15th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 23–32, Apr. 2009.

[9] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In 30th *IEEE Real-Time Systems Symposium (RTSS)*, pages 469–478, Dec. 2009.

[10] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In 21st *Euromicro Conf. on Real-Time Sys. (ECRTS)*, pages 239–248, Jul. 2009.

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

[12] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In 14th *International Conference On Principles Of Distributed Systems (OPODIS)*, Dec. 2010.

[13] A. Wieder and B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In 8th *IEEE International Symposium on Industrial Embedded Systems (SIES'13)*, Jun. 2013.