Blocking-Aware Partitioning for Multiprocessors*

Farhang Nemati, Thomas Nolte and Moris Behnam
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
{farhang.nemati, thomas.nolte, moris.behnam}@mdh.se

***Abstract*—In the multi-core and multiprocessor domain there are two scheduling approaches, global and partitioned scheduling. Under global scheduling each task can execute on any processor while under partitioned scheduling tasks are allocated to processors and migration of tasks among processors is not allowed. Under global scheduling the higher utilization bound can be achieved, but in practice the overheads of migrating tasks is high. On the other hand, besides simplicity and efficiency of partitioned scheduling protocols, existing scheduling and synchronization methods developed for uniprocessor platforms can more easily be extended to partitioned scheduling. This also simplifies migration of existing systems to multi-cores. An important issue related to partitioned scheduling is how to distribute tasks among processors/cores to increase performance offered by the platform. However, existing methods mostly assume independent tasks while in practice a typical real-time system contains tasks that share resources and they may block each other. In this paper we propose a blocking-aware partitioning algorithm to distribute tasks onto different processors. The proposed algorithm allocates a task set onto processors in a way that blocking times of tasks are decreased. This reduces the total utilization which has the potential to decrease the total number of needed processors/cores.**

I. INTRODUCTION

Multi-core (single chip multiprocessor) is today the dominating technology for desktop computing and the performance of using multiprocessors depend on the nature of the applications as well as the implementation of the software. To take advantage of the concurrency offered by a multi-core architecture, appropriate algorithms have to be used to divide the software into tasks (threads) and distribute tasks on cores to increase the performance. Real-time systems can highly benefit from multi-core processors, as critical functionality can have dedicated cores and independent tasks can run concurrently to improve performance and thereby enable new functionality. Moreover, since the cores are located on the same chip and typically have shared memory, communication between cores is very fast. Since embedded real-time systems are typically multi threaded, they are easier to adapt to multi-core than single-threaded, sequential programs. If the tasks are independent, it is a matter of deciding on which core each task should execute. For embedded real-time systems, practically, a static and manual assignment of processors is often preferred for predictability reasons.

There are two approaches for scheduling task systems on multiprocessors systems [1, 3, 7, 8]; *global* and *partitioned* scheduling. Under global scheduling, e.g., *Global Earliest Deadline First* (G-EDF), tasks are scheduled by a single scheduler based on their priorities and each task can be executed on any core. A single global queue is used for storing jobs. A task as well as a job can be preempted on a core and resumed on another core (migration of tasks among cores is permitted).

Under partitioned scheduling tasks are statically assigned to processors and tasks within each processor are scheduled by uniprocessor scheduling protocols, e.g., *Rate Monotonic* (RM) and EDF. Each processor is associated with a separate ready queue for scheduling task jobs.

However there are systems in which some tasks cannot migrate among cores while other tasks can migrate. For such systems neither of global or partitioned scheduling methods can be used. A two-level hybrid scheduling [8] which is a mix of global and partitioned scheduling methods is used for those systems.

Partitioned scheduling protocols have been used more often and are supported (with fixed priority scheduling) widely by commercial real-time operating systems [13], because of their simplicity, efficiency and predictability. However, *partitioning,* which allocates tasks to processors, is a bin-packing problem which is known to be a NP-hard problem in the strong sense; therefore finding an optimal solution in polynomial time is not realistic in the general case. Thus heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been studied to find a near-optimal partitioning [1, 7].

While in real applications tasks often share resources, many of the scheduling protocols and existing partitioning algorithms for multiprocessors (multi-cores) assume independent tasks.

A. Contributions

The first contribution of this paper is to propose a blocking-aware heuristic algorithm to allocate tasks onto the processors of a single chip multiprocessor (multi-core) platform. The algorithm extends a bin-packing algorithm with synchronization parameters. The second contribution is to implement and evaluate the algorithm and compare it to the blocking-agnostic bin-packing partitioning algorithm. Blocking-agnostic algorithm, in the context of this paper refers to a bin-packing algorithm that does not consider blocking parameters to increase the performance of partitioning, although blocking times are included in the schedulability test. The new algorithm identifies task

* This work was partially supported by the Swedish Foundation for Strategic Research (SSF) via the strategic research centre (PROGRESS) at Mälardalen University.

constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, and extends the *best-fit decreasing* (BFD) bin-packing algorithm with blocking time parameters. The objective of the algorithm is to decrease blocking overheads by assigning tasks to appropriate processors (partitions).

In practice, industrial systems mostly use Fixed Priority Scheduling (FPS) protocols. To our knowledge the only synchronization protocol under fixed priority partitioned scheduling, for multiprocessor platforms is *Multiprocessor Priority Ceiling Protocol* (MPCP) which was proposed by Rajkumar in [19]. Our algorithm assumes that MPCP is used for lock-based synchronization. Hence, we will discuss this protocol in more details in Section III.

The rest of the paper is as follows: we present the task and platform model in Section II, describe the MPCP in Section III. We present the partitioning algorithm in Section IV. In Section V the experimental results of our algorithm are presented and the results are compared to the blocking-agnostic BFD.

B. Related Work

A study of bin-packing algorithms for designing distributed real-time systems is presented in [18]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes. However, their partitioning method assumes independent tasks.

Liu et al. [14] present a heuristic algorithm for allocating tasks in multi-core-based massively parallel systems. Their algorithm has two rounds; in the first round processes (groups of threads - partitions in this paper) are assigned to processing nodes, and the second round allocates tasks in a process to the cores of a processor. However, the algorithm does not consider synchronization between tasks.

Baruah and Fisher have presented a bin-packing partitioning algorithm (*first-fit decreasing* (FFD) algorithm) in [3] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm assigns each task τ_i to the first processor, P_k for which both of following conditions, under the Earliest Deadline First (EDF) scheduling, hold:

$$D_i - \sum_{\tau_j \in P_k} DBF^*(\tau_j, D_i) \geq C_i$$

and

$$1 - \sum_{\tau_j \in P_k} u_j \geq u_i$$

where C_i, D_i and T_i specify worst-case execution time (WCET), deadline and period of task τ_i respectively, $u_i = \frac{C_i}{T_i}$, and

$$DBF^*(\tau_i, t) = \begin{cases} 0, & if\ t < D_i \\ C_i + u_i \times (t - D_i), & \text{otherwise} \end{cases}$$

The algorithm, however, assumes that tasks are independent while in practice tasks often share resources and therefore blocking time overheads must be considered while schedulability of tasks assigned to the a processor is checked. Our algorithm not only considers resource sharing when distributing tasks but it tries to reduce blocking times as well. On the other hand their algorithm works under the EDF scheduling protocol while most existing real-time systems use fixed priority scheduling policies. Our proposed algorithm works under fixed priority scheduling protocols, although it can easily be extended to other policies.

Of great relevance to our work is the work presented by Lakshmanan et al. in [13]. In the paper they investigate and analyze two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. They have developed a blocking-aware task allocation algorithm (an extension to BFD) and evaluated it under both execution control policies.

In their partitioning algorithm, the tasks that directly or indirectly share resources are put into what they call bundles (we call them macrotasks) and each bundle is tried to be allocated onto a processor. The bundles that can not fit into any existing processors are ordered by their cost, which is the blocking overhead that they introduce into the system. Then the bundle with minimum cost is broken and the algorithm is run from the beginning. However, their algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its size (utilization). Furthermore, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle is considered which could help to allocate tasks from a broken bundle to appropriate processors to decreases the blocking times.

However, their experimental results show that a blocking-aware bin-packing algorithm for suspend-based execution control policy does not have significant benefits compared to a blocking-agnostic bin-packing algorithm. Firstly, for the comparison, they have only focused on the processor reduction issue; they suppose that the algorithm is better if it reduces the number of processors. In this perspective they claim that in the worst case the number of needed processors would be equal to the number of tasks, while the worst case could be the case that an algorithm fails to schedule a task set. In our experimental evaluation, besides processor reduction, we have considered this issue as well. If an algorithm can schedule some task sets while others fail, we consider it as a benefit. Secondly, in their experiments they have not investigated the effect of some parameters such as the different number of resources, variation in the number and length of critical sections of tasks. By considering these parameters, our experimental results show that in most cases our blocking-aware algorithm has significantly better results than blocking-agnostic algorithms.

In the context of multiprocessor synchronization, the first protocol was MPCP presented by Rajkumar in [18],

which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Our partitioning algorithm attempts to decrease blocking times under MPCP and consequently decrease worst case response times which in turn may reduce the number of needed processors. Gai et al. [11, 12] present MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors. The shared resources are classified as either (i) local resources that are shared among tasks assigned to the same processor, or (ii) global resources that are shared by tasks assigned to different processors. In MSRP, tasks synchronize local resources using SRP [2], and access to global resources is guaranteed a bounded blocking time. Lopez et al. [15] present an implementation of SRP under P-EDF. Devi et al. [9] present a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short, simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is on soft real-time systems.

Block et al. [4] present FMLP (Flexible Multiprocessor Locking Protocol), which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [5]. However, to our knowledge there is no schedulability test for FMLP.

Recently, a synchronization protocol under fixed priority scheduling, has been proposed by Easwaran et al. in [10], but they focus on a global scheduling approach.

II. TASK AND PLATFORM MODEL

We will assume a task set that consists of *n* sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{c_{i,p,q}\})$ where T_i is the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. The tasks share a set of resources, R which are protected using semaphores. The set of critical sections, in which task τ_i requests resources in R is denoted by $\{c_{i,p,q}\}$, where $c_{i,p,q}$ indicates the maximum execution time of the p^{th} critical section of task τ_i in which the task locks resource $R_q \in R$. Critical sections of tasks should be sequential or properly nested. The deadline of each job is equal to T_i. A job of task τ_i, is specified by J_i .The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

We will also assume that the multiprocessor (multi-core) platform is composed of identical, unit-capacity processors (cores) with shared memory. The task set is partitioned into partitions $\{P_1, \dots, P_m\}$, and each partition is allocated onto one processor (core), thus *m* represent the minimum number of processors needed.

III. THE MPCP-MULTIPROCESSOR PRIORITY CEILING PROTOCOL

A. Definition

The MPCP is used for synchronizing a set of tasks sharing lock-based resources under a partitioned FPS protocol, i.e., RM. Under MPCP, resources are divided into *local* and *global* resources. Local resources are shared only among tasks from the same processor and global resources are shared by tasks assigned to different processors. The local resources are protected using a uniprocessor synchronization protocol, i.e., Priority Ceiling Protocol (PCP) [20]. A task blocked on a global resource suspends and makes the processor available for the local tasks. A critical section in which a task performs a request for a global resource is called *global critical sections* (*gcs*). Similarly a critical section where a task requests for a local resource is denoted as *local critical sections* (*lcs*).

The blocking time of a task in addition to local blocking, needs to include *remote blocking* where a task is blocked by tasks (with any priority) executing on other processors. However, the maximum remote blocking time of a job is bounded and is a function of the duration of critical sections of other jobs. This is a consequence of assigning any *gcs* a ceiling greater than the priority of any other task, hence a *gcs* can only be blocked by another *gcs* and not by any non-critical section. Assume ρ_H is the highest priority among all tasks. The priority of a job J_i executing within a *gcs* in which it requests R_k is called *remote ceiling* of *gcs* and equals to $\rho_H + 1 + \max\{\rho_j | \tau_j \; requests \; R_k \; and \; \tau_j \; is \; not \; on \; J_i\text{'s processor}\}$.

Global critical sections cannot be nested in local critical sections and vice versa. Global resources potentially lead to high blocking times, thus tasks sharing the same resources are preferred to be assigned to the same processor as far as possible. Our proposed algorithms attempt to reduce the blocking times by assigning tasks to appropriate processors.

To determine the schedulability of each processor under RM scheduling the following test is performed:

$$\forall k \; 1 \le i \le n, \sum_{k=1}^{i} C_k/T_k + B_i/T_i \le i\left(2^{1/i} - 1\right) \quad (1)$$

where n is the number of tasks assigned to the processor, and B_i is the maximum blocking time of task τ_i which includes remote blocking factors as well as local blocking time. However this condition is sufficient but not necessary. Thus for more precise schedulability test of tasks our algorithm performs response time analysis [6].

B. Blocking times of tasks

Before explaining the blocking factors of the blocking time of a job, we have to explain the following terminology:

- n_i^G: The number of global critical sections of task τ_i.
- $\{J'_{i,r}\}$: The set of jobs on processor P_r (other than J_i's processor) with global critical sections having priority higher than the global critical sections of jobs that can directly block J_i.
- $NH_{i,r,k}$: The number of global critical sections of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.

- $\{GR_{i,k}\}$: The set of global resources that will be locked by both J_i and J_k.
- $NC_{i,k}$: The number of global critical sections of J_k in which it request a global resource in $\{GR_{i,k}\}$.
- ${\beta_i}^{local}$: The longest local critical section among jobs with a priority lower than that of job J_i executing on the same processor as J_i which can block J_i.
- ${\beta L_i}^{global}$: The longest global critical section of any job J_k with a priority lower than that of job J_i executing on a different processor than J_i's processor in which J_k requests a resource in $\{GR_{i,k}\}$.
- ${\beta H_{i,k}}^{global}$: The longest global critical section of job J_k with a priority higher than that of job J_i executing on a different processor than J_i's processor. In this global critical section, J_k requests a resource in $\{GR_{i,k}\}$.
- ${\beta'_{i,k}}^{global}$: The longest global critical section of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- ${\beta_{i,k}}^{lg}$: The longest global critical section of a lower priority job J_k on the J_i's host processor.

The maximum blocking time B_i of task τ_i is a summation of five blocking factors:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} + B_{i,5}$$

where:

1. $B_{i,1} = n_i^G {\beta_i}^{local}$ each time job J_i is blocked on a global resource and suspends the local lower priority jobs may execute and lock local resources and block J_i when it resumes.
2. $B_{i,2} = n_i^G {\beta L_i}^{global}$ when a job J_i is blocked on a global resource which is locked by a lower priority job executing on another processor.
3. $B_{i,3} = \sum_{\substack{\rho_i \le \rho_k \text{ and} \\ J_k \text{is not on } J_i\text{'s processor}}} NC_{i,k} \lceil T_i/T_k \rceil {\beta H_{i,k}}^{global}$
   when higher priority jobs on processors other than J_i's processor block J_i.
4. $B_{i,4} = \sum_{\substack{J_k \in \{J'_{i,r}\} \text{ and} \\ P_r \neq J_i\text{'s processor}}} NH_{i,r,k} \lceil T_i/T_k \rceil {\beta'_{i,k}}^{global}$
   when the *gcs*'s of lower priority jobs on processor P_r (different from J_i's processor) are preempted by higher priority *gcs*'s of $J_k \in \{J'_{i,r}\}$.
5. $B_{i,5} = \sum_{\substack{\rho_k \le \rho_i \text{ and} \\ J_k \text{ on } J_i\text{'s processor}}} min\,(n_i^G + 1, n_k^G)\, {\beta_{i,k}}^{lg}$
   when J_i is blocked on global resources and suspends a local job J_k can execute and enter a global section which can preempt J_i when it executes in non-*gcs* sections.

IV. PARTITIONING ALGORITHM

In this section we present a partitioning algorithm that groups tasks into partitions so that each partition can be allocated and scheduled on one processor. The objective of the algorithm is to decrease the blocking times of tasks. This generally increases the schedulability of a task set which may reduce the number of partitions (processors).

Considering the blocking factors of tasks under MPCP, tasks with more and longer global critical sections lead to more blocking times. This is also shown by experiments presented in [11]. Our goal is to (i) decrease the number of global critical sections by assigning the tasks sharing resources to the same partition as far as possible, (ii) decrease the ratio and time of holding global resources by assigning the tasks that request the resources more often and hold them longer to the same partition as long as possible.

In our previous work [16, 17] we have proposed a partitioning algorithm in which tasks are grouped together based on task preferences and constraints. The algorithm partitions tasks based on a cost function which is derived from task preferences and constraints. In [16] the resource sharing is only local by means of allocating the tasks that directly or indirectly share resources onto the same processor. Tasks that directly or indirectly share resources are called *macrotasks*, e.g. if tasks τ_i and τ_j share resource R_p and tasks τ_j and τ_k share resource R_q, all three tasks belong to the same macrotask. However if a macrotask does not fit in one processor (is not schedulable) the algorithm fails. In [17] tasks belonging to the same macrotask can be allocated to different partitions (processors), thus it is more flexible but it introduces remote blocking overhead into the systems. The goal of the algorithm is to put the tasks into appropriate partitions so that the costs are minimized. The algorithm may have different partitioning strategies, e.g., increasing cash hits, decreasing blocking times, etc. The strategy of partitioning may differ, depending on the nature of a system, and result in different partitions. In current work, however, we focus on decreasing remote blocking overheads of tasks which leads to increasing the schedulability of a task set and possibly reducing the number of processors needed for scheduling the task set.

We have developed a blocking-aware algorithm that is an extension to the BFD algorithm. In a blocking-agnostic BFD algorithm, bins (processors) are ordered in non-increasing order of their utilization and tasks are ordered in non-increasing order of their size (utilization). The algorithm tries to allocate the task from the top of the ordered task set onto the first processor that fits it, beginning from the top of the ordered processor list. If none of the processors can fit the task, a new processor is added to the processor list. At each step the schedulability of all processors should be tested, because allocating a task to a processor can increase the remote blocking time of tasks previously allocated to other processors and may make the other processors unschedulable. This means that it is possible that even if a task is allocated to a new processor, some of the previous processors become unschedulable which makes the algorithm fail.

A. The Algorithm

The algorithm performs partitioning of a task set in two parallel alternatives and the result will be the output of the alternative with better partitioning results. However, the algorithm performs a few common steps before starting to perform the parallel alternatives. Each alternative allocates tasks to the processors (partitions) in a different strategy. When a bin-packing algorithm allocates an object (task) to a bin (processor), it usually puts the object in a bin that fits it better, and it does not consider the unallocated objects that will be allocated after the current object. However, the first alternative of our algorithm considers the tasks that are not allocated to any processor yet; and tries to take as many as possible of the best related tasks (based on remote blocking parameters) with the current task. On the other hand, the second alternative considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. The second alternative performs more like the usual bin-packing algorithms, although it considers the remote blocking parameters while allocating a task to a processor.

The common steps of the algorithm before the two alternatives are performed in parallel are as follows.

1. Each task is assigned a weight. The weight of each task, besides its utilization, depends on parameters that lead to potential remote blocking time caused by other tasks:

$$w_i = \left\lceil (C_i + \sum_{\rho_i<\rho_k} NC_{i,k}\,\beta_{i,k}\lceil T_i/T_k\rceil + NC_i\, max_{\rho_i\geq\rho_k}\beta_{i,k})/T_i \right\rceil \quad (2)$$

where, $\beta_{i,k}$ is the longest critical section of task τ_k in which it shares a resource with τ_i, and NC_i is the total number of critical sections of τ_i.

2. Macrotasks are generated; the tasks that directly or indirectly share resources are put into the same macrotask. A macrotask has two alternatives; it can either be broken or unbroken. If a macrotask cannot fit (cannot be scheduled) in one processor, it is assigned as broken, otherwise it is denoted as unbroken. If a macrotask is unbroken, the partitioning algorithm always allocate all tasks in the macrotask to the same partition (processor). This means that all tasks in the macrotask will share resources locally relieving tasks from remote blocking. However, tasks within a broken macrotask will be distributed into more than one partition. Similar to tasks, a weight is assigned to each unbroken macrotask, which equals to the sum of weights of its tasks.
3. The unbroken macrotasks together with the tasks that do not belong to any unbroken macrotasks are ordered in a single list in non-increasing order of their weights. We call this list the *mixed list*.

The strategy of allocation of tasks in both alternatives depends on attraction between tasks. The attraction of task τ_k to a task τ_i is defined based on the potential remote blocking overhead that task τ_k can introduce to task τ_i if they are allocated onto different processors. We represent the attraction of task τ_k to task τ_i as $v_{i,k}$ which is defined as follows:

$$v_{i,k} = \begin{cases} \mathrm{NC_{i,k}}\beta_{i,k}\lceil T_i/T_k\rceil, & \rho_i < \rho_k \\ \mathrm{NC_i}\beta_{i,k}, & \rho_i \geq \rho_k \end{cases} \quad (3)$$

Now we present the continuation of each alternative separately.

Alternative 1:

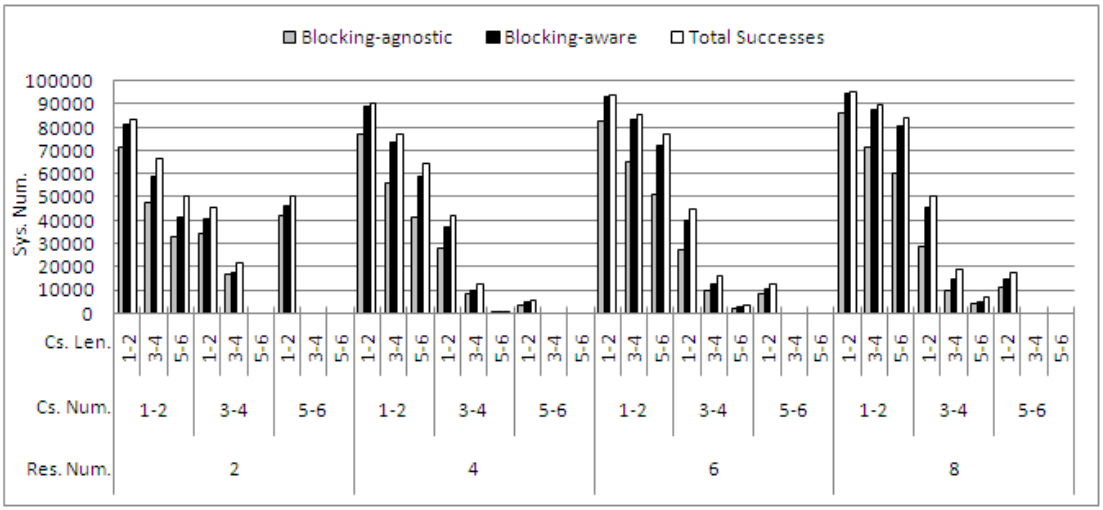
After step 3 the following steps are repeated by alternative 1 until all tasks are allocated to processors (partitions):

4. All processors are ordered in their non-increasing order of utilization.
5. The object at the top of the mixed list is picked.
   a. If the object is a task and it does not belong to a broken macrotask it will be allocated onto the first processor that fits it, beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit the task a new processor is added to the list and the task is allocated onto it. In this case if one or more of the processors becomes unschedulable this alternative of the algorithm fails.
   b. If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits them. If none of the processors can fit the tasks, they will be allocated onto a new processor and in this case, if one or more of the processors becomes unschedulable the Alternative 1 fails.
   c. If the object is a task that belongs to a broken macrotask, the algorithm orders the tasks (those that are not allocated yet) within the macrotask in non-increasing order of attraction to the task based on equation (3). We call this list the *attraction list* of the task. The task itself will be on the top of its attraction list. The best processor for allocation is selected, which is the processor that fits the most tasks from the attraction list, beginning from the top of the list. If none of the existing processors can fit any of the tasks, a new processor is added and as many tasks as possible from the attraction list are allocated to the processor. However, if the new processor cannot fit any task from the attraction list, i.e., one or more of the processors become unschedulable, the Alternative 1 fails.

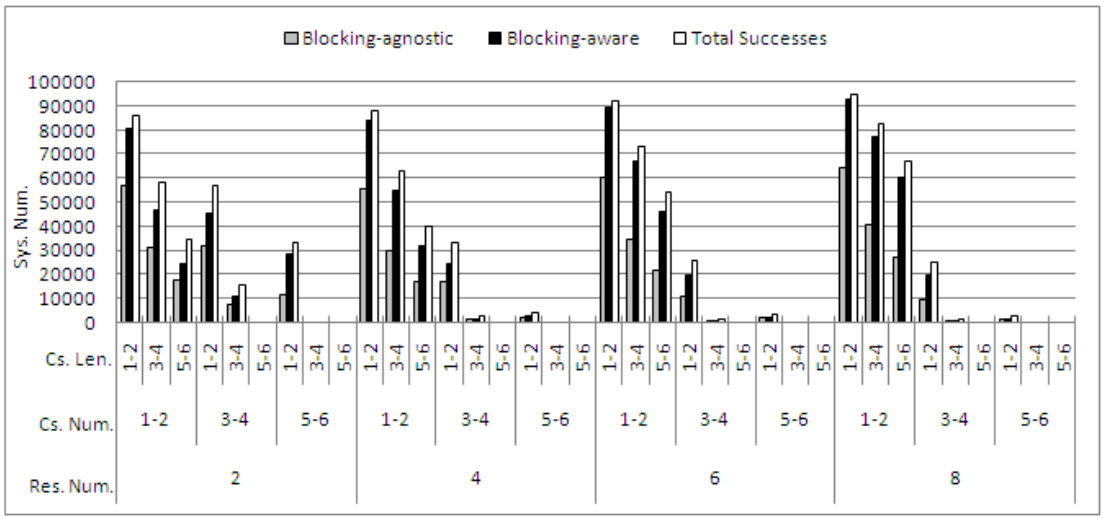
Alternative 2:

The following steps are repeated until all tasks are allocated to processors:

1. The object at the top of the mixed list is picked.
   a. If the object is a task and it does not belong to a broken macrotask it will be allocated onto the first processor that fits it, beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit the task a new processor is added to the list and the task is



(a) 3 tasks per processor



(b) 6 tasks per processor

Figure 1. Total number of task sets that the algorithms successfully schedule (task sets generated from 3 fully utilized processors).

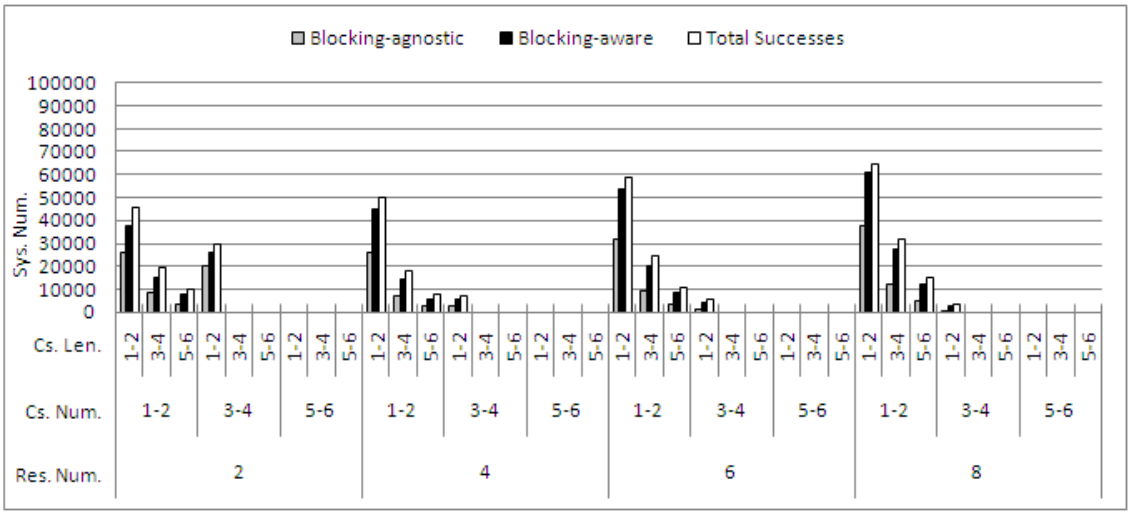
allocated onto it. In this case if one or more of the processors becomes unschedulable the Alternative 2 fails.

b. If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits them. If none of the processors can fit the tasks, they will be allocated onto a new processor and in this case, if one or more of the processors becomes unschedulable the Alternative 2 fails.
c. If the object is a task that belongs to a broken macrotask, the ordered list of processors is a concatenation of two ordered lists of processors. The top list contains the processors that include some tasks from the macrotask of the task; this list is ordered in non-increasing order of processors' attraction to the task based on equation (3), i.e. the processor which has the greatest sum of attractions of its tasks to the picked task is the most attracted processor to the task. The second list of processors is the list of those processors that do not contain any task from the macrotask of the picked task and are ordered in non-increasing order of their utilization. The picked task will be allocated onto the first processor from the processor list that will fit it. The task will be allocated to a new processor if none of the existing ones can fit it. And this alternative of the algorithm fails if allocating the task to the new processor makes some of the processors unschedulabe.

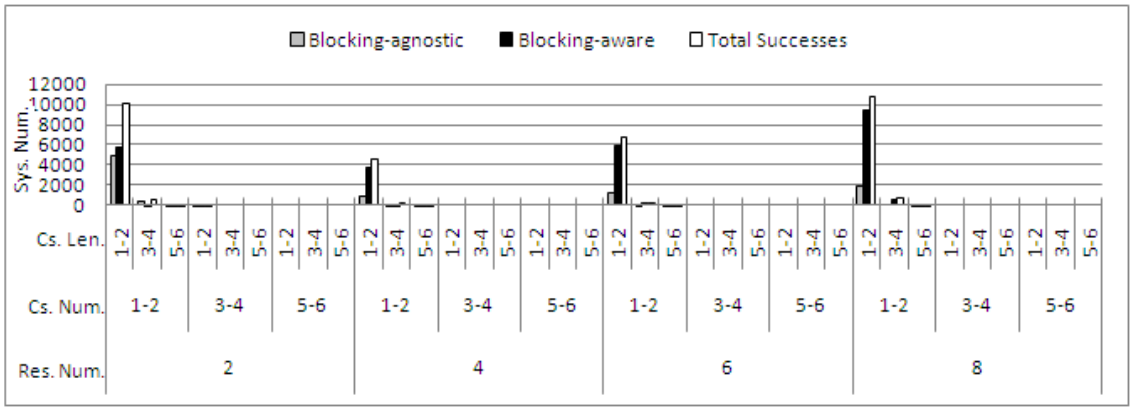
The algorithm fails if both alternatives fail to schedule a task set. If one of the alternatives fails the result will be the output of the other one. Finally if both succeed to schedule the task set, the one with less partitions (processors) will be the output of the algorithm.

V. EXPERIMENTAL EVALUATION

In this section we present our experimental results of the blocking-aware bin-packing algorithm together with the blocking-agnostic algorithm. For a number of systems (task sets), we have compared the performance of the algorithms in two different aspects; 1) The total number of systems that each of the algorithms can schedule, 2) The



(a) Workload: 6 fully utilized processors, 3 tasks per processor



(b) Workload: 8 fully utilized processors, 6 tasks per processor

Figure 2. Total number of task sets that the algorithms successfully schedule

total number of systems that one of the algorithms schedules with fewer processors when both succeed.

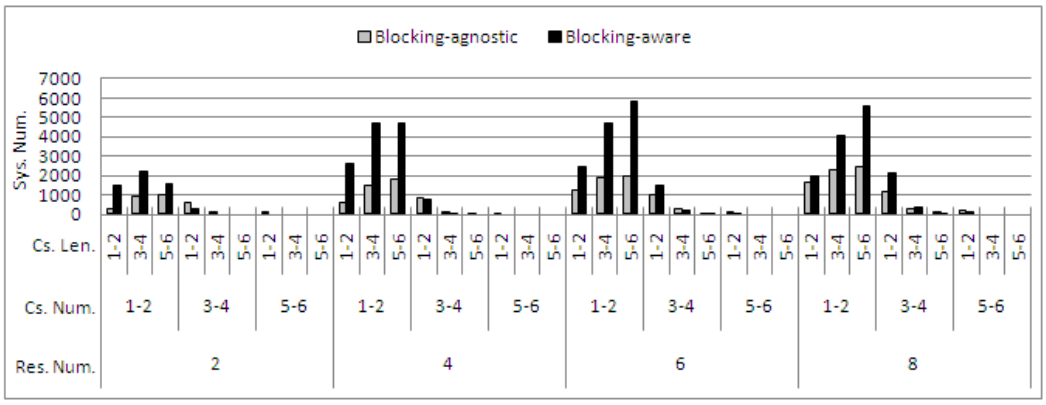
A. Task Set Generation

We generated systems (task sets) for different workloads; we denote workload as a defined number of fully utilized processors. Given a workload, the full capacity of each processor (utilization of 1) is randomly divided into a defined number of tasks utilizations. Usually for generating systems, utilization and periods are randomly assigned to tasks and worst case execution times of tasks are calculated based on them. However, in our system generation, the worst case execution times (WCET) of tasks are randomly assigned and the period of each task is calculated based on its utilization and WCET. The reason is that we had to restrict that the WCET of a task not to be less than the total length of its critical sections. Since we have limited the maximum number of critical sections to 10 and the maximum length of any critical section to 10 time units, hence the WCET of each task should be greater than 100 (10×10) time units. The WCET of each task was randomly chosen between 100 and 150 time units. The system generation was based on different settings; the input parameters for settings are as follows.

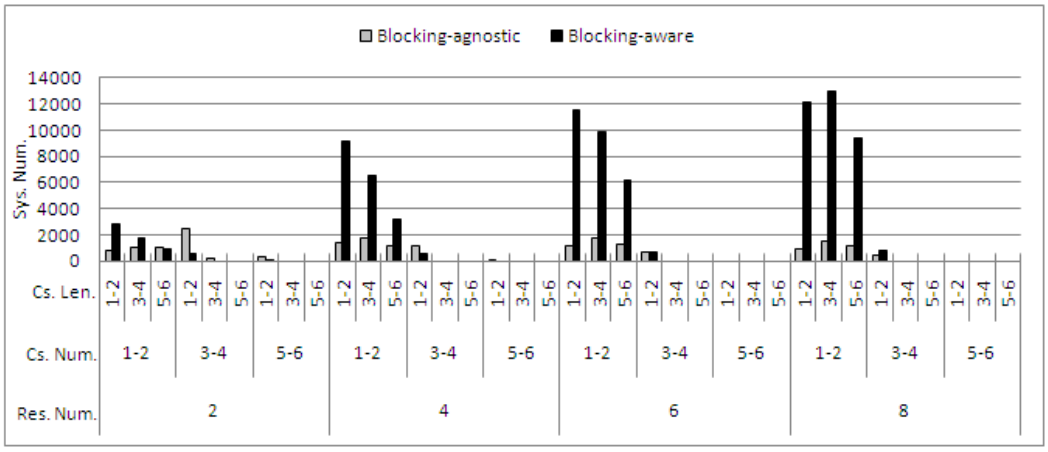
1. Workload (3, 4, 6, or 8 fully utilized processors),
2. The number of tasks per processor (3 or 6 tasks per processor),
3. The number of resources (2, 4, 6, or 8),
4. The range of the number of critical sections per task (1 to 2, 3 to 4 or 5 to 6 critical sections per task),
5. The range of length of critical sections (1 to 2, 3 to 4, or 5 to 6).

For each setting, we generated 100.000 systems, and combining the parameters of settings (288 different settings), the total number of systems generated for the experiment were 28.800.000.

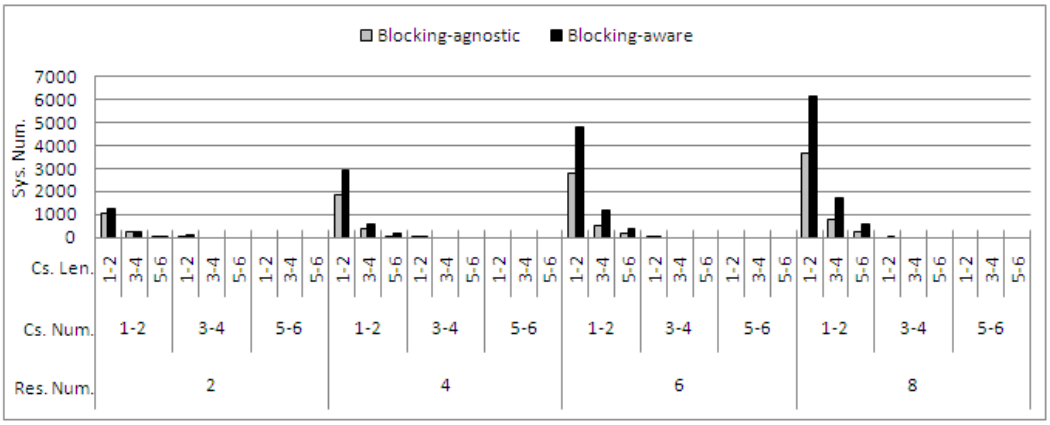
With the generated systems we were able to evaluate our partitioning algorithm with respect to different factors, i.e., various workloads (number of fully utilized processors), number of tasks per processor, number of shared resources, number of critical sections per task, and length of critical sections.



(a) 3 fully utilized processors, 3 tasks per processor



(b) 3 fully utilized processors, 6 tasks per processor



(c) 6 fully utilized processors, 3 tasks per processor

Figure 3. Total number of task sets that either of algorithms schedule with fewer processors than the other.

B. Results

In this section we present the evaluation results of our blocking-aware algorithm. We compare them to the results of the blocking-agnostic bin-packing algorithm.

The first aspect of comparison of the results from the two algorithms is the total number of systems that each algorithm succeeds to schedule. Comparison for 3 fully utilized processors is represented in Figure 1. Figures 1.a and 1.b represent the results for 3 task per processor and 6 tasks per processors respectively. The vertical axis shows the total number of systems that the algorithms could schedule successfully. The horizontal axis shows three factors in three different lines; the bottom line shows the number of shared resources within systems (Res. Num.), the second line shows the number of critical sections per task (Cs. Num.), and the top line represents the length of critical sections within each task (Cs. Len.), e.g., Res. Num.=4, Cs. Num.=1-2, and Cs. Len.=1-2 represents the systems that share 4 resources, the number of critical sections are between 1 and 2, and the length of these critical sections are between 1 and 2.

As depicted in Figure 1, considering the total number of systems that each algorithm succeeds to schedule, our blocking-aware algorithm performs better (more systems

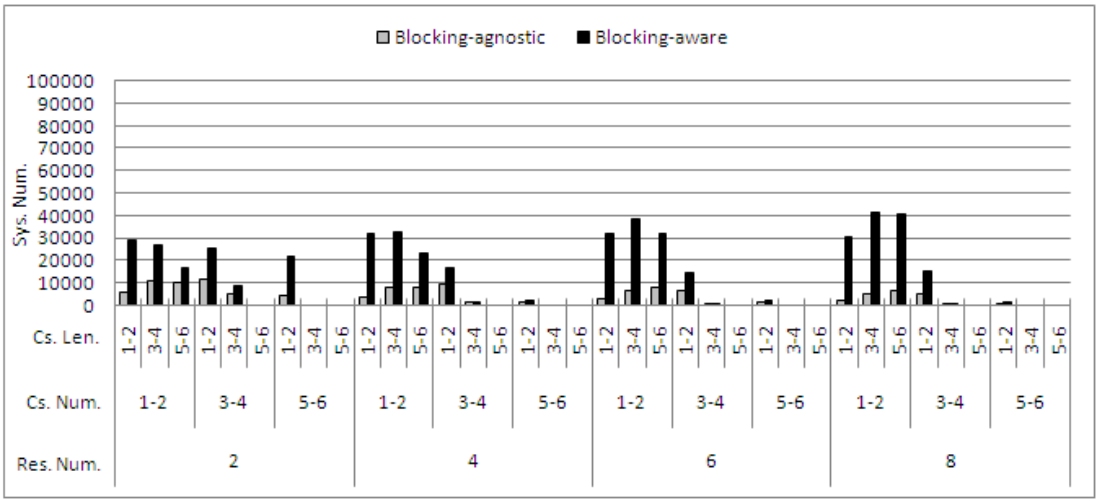


Figure 4. Total number of task sets that the algorithms exclusively schedule successfully (workload of 3 fully utilized processors, 6 tasks per processor).

are schedulable) compared to the blocking-agnostic algorithm. By increasing the number of resources, the number of successfully scheduled systems in both algorithms is slightly increased. The reason for this behavior is that with fewer resources, more tasks share the same resource introducing more blocking overheads which leads to fewer schedulable systems. However, it is shown that the blocking-aware algorithm performs better as the number of resources is increased. It is also shown that increasing the number and/or the length of critical sections significantly reduces the number of schedulable systems in both algorithms. As the number of tasks per processor is increased from 3 (Figure 1.a) to 6 (Figure 1.b), the blocking-aware algorithm performs significantly better (schedules more systems) than the blocking-agnostic algorithm.

As the workload (the number of fully utilized processors) is increased, although the blocking-aware algorithm still performs better than the blocking-agnostic algorithm, the number of schedulable systems by both algorithms is reduced (Figure 2). The reason for this behavior is that the number of tasks within systems are relatively many (48 tasks per each system in Figure 2.b) and the workload is high (8 fully utilized processors in Figure 2.b), and all the tasks within systems share resources. This introduces a lot of interdependencies among tasks and consequently a huge amount of blocking overheads, making fewer systems schedulable. In practice in big systems with many tasks, not all of the tasks share resources, which leads to fewer interdependencies among tasks and less blocking times. However, we continued the experiment with higher workload in the same way as the other experiments (that all tasks share resources) to be able to compare the results with the previous results. We believe that realistic systems, even with high workload and many tasks can significantly benefit from our partitioning algorithm to increase the performance.

The second aspect for comparison of performance of the algorithms is the total number of systems that each algorithm schedules with fewer processors than the other one (better in processor reduction). The results show (Figure 3) that our blocking-aware algorithm mostly performs significantly better than the blocking-agnostic bin-packing algorithm, especially given a lower number of critical sections per task and shorter critical sections. However, for lower number of shared resources (e.g., 2 shared resources) especially for higher workloads (e.g., 6 fully utilized processors) the blocking-aware algorithm does not always perform better (Figure 3).

In the experiment we also studied the results of both alternatives of the algorithm separately. The results show that the alternative 1 mostly performs significantly better than the alternative 2, although in some cases the alternative 2 performs better especially as the number and the length of critical sections increase.

C. *Combination of Algorithms*

The results in Section V.B. show that our blocking-aware partitioning algorithm mostly performs significantly better in both increasing the number of schedulable systems as well as processor reduction. However, finding an optimal solution with a bin-packing algorithm is not realistic (bin-packing is a NP-hard problem in the strong sense), hence there may exist schedulable systems that our algorithm fails to schedule. As illustrated in Figure 4, the number of systems that our algorithm can exclusively schedule (i.e., the blocking-agnostic algorithm fails to schedule them) are significantly higher compared to the number of systems exclusively schedulable by the blocking-agnostic algorithm. However, there are still some systems that are only schedulable by the blocking-agnostic algorithm. Thus, combination of both algorithms can be convenient to improve the overall results. It can also be noticed in Figure 1 that combining the results of both algorithms leads to more schedulable systems (Total successes). Furthermore, as shown in Figure 2, there are some systems that the blocking-agnostic algorithm schedules with fewer processors (it performs better in processor reduction). Hence a combined approach will lead to an improvement in processor reduction as well.

VI. SUMMARY AND FUTURE WORK

In this paper we have proposed a heuristic blocking-aware algorithm, for real-time multiprocessor systems, which extends a bin-packing algorithm with synchronization parameters. The algorithm allocates a task set onto the processors of a single-chip multiprocessor (multi-core) with shared memory. The objective of the algorithm is to decrease blocking times of tasks by means of allocating the tasks that directly or indirectly share resources onto appropriate processors. This generally increases shedulability of a task set and can lead to fewer processors compared to blocking-agnostic bin-packing algorithms.

Since in practice most systems use fixed priority scheduling protocols, we have developed our algorithm under MPCP, the only existing synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling. This protocol introduces large amounts of blocking time overheads especially when the global resources are relatively long and the access ratio to them is high.

Our experimental results confirm that our algorithm mostly performs significantly better with respect to system schedulability and processor reduction. However, given a NP-hard problem, a bin-packing algorithm may not achieve the optimal solution, i.e., our results show that, although our algorithm mostly performs significantly better, there still exist some cases that can only be solved by the blocking-agnostic approach. Thus we show that a combination of both algorithms improves the results with respect to the total number of schedulable systems and processor reduction.

A future work will be extending our partitioning algorithm to other synchronization protocols, e.g., MSRP and FMLP for partitioned scheduling. Another interesting future work is to apply our approach to real systems and study the performance gained by the algorithm on these systems. In the domain of multiprocessor scheduling and synchronization our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

REFERENCES

[1] T. Baker. A Comparison of Global and Partitioned EDF Schedulability Test for Multiprocessors. *Technical Report TR-051101*, Department of Computer Science, Florida State University, 2005.

[2] T. Baker. Stack-based Scheduling of Real-time Processes. *J.Real-Time Systems, vol. 3, no. 1, pp. 67-99,* 1991.

[3] S. Baruah, and N. Fisher. The Partitioned Multiprocessor Scheduling of Sporadic Task Systems. *In proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05), pp. 321- 329,* 2005.

[4] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A Flexible Real-time Locking Protocol for Multiprocessors. *In 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07), pp. 47-56*, 2007.

[5] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on Multiprocessors: To Block or not to Block, to Suspend or Spin? *In proceedings of 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08), pp. 342-353, 2008.*

[6] A. Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. *In S. Son, editor, Advances in Real-Time Systems, pp. 225-248*, *Prentice-Hall*, 1994.

[7] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. *In J. Y. Leung, editor, Handbook on Scheduling Algorithms, Methods, and Models, pp. 30.1-30.19. ChapmanHall/CRC*, 2004.

[8] U. Devi. Soft Real-Time Scheduling on Multiprocessors. *PhD thesis*, *www.cs.unc.edu/˜anderson/diss/devidiss.pdf.* 2006.

[9] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. *In proceedings of 18th IEEE Euromicro Conference on Real-time Systems (ECRTS'06), pp. 75-84*, 2006.

[10] A. Easwaran, B. Andersson. Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling. *In Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09),* 2009.

[11] 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 Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, *pp. 189-198*, 2003.

[12] P. Gai, G. Lipari, andM. 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 (RTSS'01)*, *pp. 73-83,* 2001.

[13] 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),* 2009.

[14] Y. Liu, X. Zhang, H. Li, and D. Qian. Allocating Tasks in Multi-core Processor Based Parallel Systems. *Network and Parallel Computing Workshops, in conjunction with IFIP'07, pp. 748-753*, 2007.

[15] J. M. López , J. L. Díaz , and D. F. García. Utilization Bounds for EDF Scheduling on Real-time Multiprocessor Systems. *Real-Time Systems, v.28 n.1, pp. 39-68,* 2004.

[16] F. Nemati, M. Behnam, and T. Nolte. Multiprocessor Synchronization and Hierarchical Scheduling. *In Proceedings of First Intl. Workshop on Real-time Systems on Multicore Platforms: Theory and Practice (XRTS-2009) in conjunction with ICPP'09*, 2009.

[17] F. Nemati, M. Behnam, and T. Nolte. Efficiently Migrating Real-Time Systems to Multi-Cores. *In Proceedings of 14th IEEE International Conference on Emerging Techonologies and Factory (ETFA'09),* Mallorca, Spain, September, 2009.

[18] D. de Niz, and R. Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems. *Intl. Journal of Embedded Systems, Vol. 2, No. 3-4, pp. 196-208*, 2006.

[19] R. Rajkumar. Synchronization in multiple processor systems. *In Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[20] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time System Synchronization. *IEEE Transactions on Computers, 39(9), pp. 1175-1185,* 1990.