Partitioning Real-Time Systems on Multiprocessors with Shared Resources*

Farhang Nemati, Thomas Nolte, and Moris Behnam

Mälardalen Real-Time Research Centre, Västerås, Sweden
{farhang.nemati,thomas.nolte,moris.behnam}@mdh.se

Abstract. In this paper we propose a blocking-aware partitioning algorithm which allocates a task set on a multiprocessor (multi-core) platform in a way that the overall amount of blocking times of tasks are decreased. The algorithm reduces the total utilization which, in turn, has the potential to decrease the total number of required processors (cores). In this paper we evaluate our algorithm and compare it with an existing similar algorithm. The comparison criteria includes both number of schedulable systems as well as processor reduction performance.

1 Introduction

Two main approaches for scheduling real-time systems on multiprocessors exist; global and partitioned scheduling [1–4]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor. A single global queue is used for storing jobs. A job can be preempted on a processor and resumed on another processor, i.e., migration of tasks among processors is permitted. Under a partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) and EDF. Each processor is associated with a separate ready queue for scheduling task jobs.

Partitioned scheduling protocols have been used more often and are supported (with fixed priority scheduling) widely by commercial real-time operating systems [5], inherent in their simplicity, efficiency and predictability. Besides, the well studied uniprocessor scheduling and synchronization methods can be reused for multiprocessors with fewer changes (or no changes). However, partitioning (allocating tasks to processors) is known to be a bin-packing problem which is a NP-hard problem in the strong sense; hence finding an optimal solution in polynomial time is not realistic in the general case. Thus, to take advantage of the performance offered by multi-cores, scheduling protocols should be coordinated with appropriate partitioning algorithms. Heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been developed to find a near-optimal partitioning [1, 3]. However, the scheduling protocols and existing partitioning algorithms for multiprocessors (multi-cores) mostly assume independent tasks while in real applications, tasks often share resources.

* This work was partially supported by the Swedish Foundation for Strategic Research (SSF) via Mälardalen Real-Time Research Centre (MRTC) at Mälardalen University.

C. Lu, T. Masuzawa, and M. Mosbah (Eds.): OPODIS 2010, LNCS 6490, pp. 253–269, 2010.
© Springer-Verlag Berlin Heidelberg 2010

We have developed a heuristic partitioning algorithm [6], under which our system assumptions include presence of mutually exclusive shared resources. The heuristic partitions a system (task set) on an identical shared memory single-chip multiprocessor platform. The objective of the algorithm is to decrease blocking overheads by assigning tasks to appropriate processors (partitions). This consequently increases the schedulability of the system and may reduce the number of processors. Our heuristic identifies task 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. In practice, industrial systems mostly use Fixed Priority Scheduling (FPS) protocols. The Multiprocessor Priority Ceiling Protocol (MPCP) which was proposed by Rajkumar in [7], for many years, has been a standard multiprocessor synchronization protocol under fixed priority partitioned scheduling. Thus, both our algorithm and an existing similar algorithm proposed in [5] assume that MPCP is used for lock-based synchronization. We have investigated MPCP in more details in [6]. Our algorith, however, can be easily extended to other synchronization protocols under partitioned scheduling policies. The algorithm proposed in [5] is named the Synchronization-Aware Partitioning Algorithm (SPA), and our algorithm is named the Blocking-Aware Partitioning Algorithm (BPA). From now on we refer them as SPA and BPA respectively.

1.1 Contributions

The contributions of this paper are threefold:
(1) We 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.
(2) We implement our algorithm together with the best known existing similar heuristic [5]. The implementation is modular in which any new partitioned scheduling and synchronization protocol as well as any new partitioning heuristic can easily be inserted.
(3) We evaluate our algorithm together with the existing heuristic and compare the two approaches to each other as well as to an blocking-agnostic bin-packing partitioning algorithm, used as reference. The 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 rest of the paper is as follows: we present the task and platform model in Section 2. We explain the existing algorithm (SPA) and present our partitioning algorithms (BPA) in Section 3. In Section 4 the experimental results of both algorithms are presented and the results are compared to each other as well as to the blocking-agnostic algorithm.

1.2 Related Work

A significant amount of work has been done in the domain of task allocation on multiprocessors and distributed systems. The emerging of multi-core architectures has increased the interest in the multiprocessor methods. However, in this paper we present the most related works to our approach.

Tindell et al. [8] describe a method called *simulated annealing* for partitioning a task set on a distributed system. The simulated annealing technique is not a heuristic solution but a global optimization method which is used to find a near-optimal solution. The important factor in simulated annealing is that it includes jumps to new solutions to be able to get a better one. The simulated annealing techniques do not include heuristics and it is usually difficult to find a good or even any feasible partitioning [9].

The *Slack Method* presented in [9] is a partitioning heuristic in which the first step is to divide the tasks into sets of communicating tasks (precedence constraint). The size of each set then is reduced based on the concept of *task slack* which is the delay a task can tolerate without missing its deadline. The second step is to map the sets of tasks onto the processors in a way to reduce the communication among processors.

A study of bin-packing algorithms for designing distributed real-time systems is presented in [10]. 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.

Baruah and Fisher have presented a bin-packing partitioning algorithm (first-fit decreasing (FFD) algorithm) in [11] 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, however, assumes independent tasks. On the other hand their algorithm has been developed under the EDF scheduling protocol while most existing real-time systems use fixed priority scheduling policies. The focus of our proposed heuristic, in this paper, is fixed priority scheduling protocols, although it can easily be extended to other policies.

Of great relevance to our work presented in this paper is the work presented by Lakshmanan et al. in [5]. 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 (in this paper we call them macrotasks) and each bundle is tried to be allocated onto a processor. The bundles that cannot 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 decrease 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. They have not considered the worst case as it 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. However, according to our experimental results, their heuristic performs slightly better than the blocking-agnostic algorithm, and our algorithm performs significantly better than both.

In the context of multiprocessor synchronization, Rajkumar et al. for the first time proposed a synchronization protocol in [12] which later [7] was called Distributed Priority Ceiling Protocol (DPCP). DPCP extends PCP to distributed systems and it can be used with shared memory multiprocessors. However, a major motivation of increasing interest in the multiprocessor methods is the emerging of multi-core platforms for which DPCP is not an appropriate synchronization protocol. Rajkumar in [7] presented MPCP, which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Considering that MPCP has been a standard multiprocessor synchronization protocol, 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. [13, 14] 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. [16] present a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is soft real-time systems.

Block et al. [17] present Flexible Multiprocessor Locking Protocol (FMLP) , 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 [18]. However, although in a longer version of [17][1], the blocking times have been calculated, but to our knowledge there is no concrete schedulability test for FMLP under global scheduling protocols. However, Brandenburg and Anderson in [19] have extended partitioned FMLP to fixed priority scheduling policy and derived a schedulability test for it. In a later work [20], the same authors have compared DPCP, MPCP and FMLP. However, as the partitioned scheduling approaches suffer from bin-packing problem, we believe to achieve a better and fair comparison of the approaches, they should be coordinated with task allocation algorithms.

Recently, Easwaran and Andersson have proposed a synchronization protocol [21] under global fixed priority scheduling protocol. In this paper, for the first time, the

[1] Available at http://www.cs.unc.edu/˜anderson/papers/rtcsa07along.pdf

authors have derived schedulability analysis of the priority inheritance protocol under global scheduling algorithms.

2 Task and Platform Model

In this paper we 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 denotes 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 pth 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 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 $\{P_1, \ldots, P_m\}$, and each partition is allocated onto one processor (core), thus m represent the minimum number of processors needed.

3 The Blocking Aware Partitioning Algorithms

3.1 Blocking-Aware Partitioning Algorithm (BPA)

In this section we propose 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 overall blocking times of tasks. This generally increases the schedulability of a task set which may reduce the number of required 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 [14]. 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 [22] we have presented a partitioning framework in which tasks are grouped together based on task preferences and constraints. The framework partitions tasks based on a cost function which is derived from task preferences and constraints. The framework attempts to allocate 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, there are cases that a macrotask cannot fit in one processor (i.e., assuming that the tasks in the macrotask are the only tasks allocated on a processor, still it can not be scheduled by the processor). In this case tasks belonging to the same macrotask can be allocated to different partitions (processors).

The goal of the framework presented in [22] is to put the tasks into appropriate partitions so that the costs are minimized. The framework may have different partitioning strategies, e.g., increasing cache hits, decreasing blocking times, etc. The strategy of partitioning may differ, depending on the nature of a system, and result in different partitions. The framework is a general partitioning approach without deeply focusing on any specific strategy and thus we have not presented any evaluation except one example. Obviously, for different partitioning strategies (e.g., increasing cache hits) the guiding heuristics as well as the implementation of the algorithm will be completely different. In current work, however, we specifically focus on a partitioning strategy for decreasing remote blocking overheads of tasks which leads to increasing the schedulability of a task set and possibly will reduce the number of processors required for scheduling the task set. We derive heuristics to specifically guide the partitioning algorithm to reduce the remote blocking times. We have also performed detailed experimental evaluation according to different resource sharing parameters.

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 attempts to allocate the task from the top of the ordered task set onto the first processor that fits it (i.e., the first processor on which the task can be allocated while all processors are schedulable), 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, it is possible that some of the previous processors become unschedulable even if a task is allocated to a new processor, which makes the algorithm fail.

The Algorithm: The algorithm performs partitioning of a task set in two rounds and the result will be the output of the round with better partitioning results. However, the algorithm performs a few common steps before starting to perform the rounds. Each round allocates tasks to the processors (partitions) in a different strategy. When a BFD 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. The rationale behind the two rounds is that the heuristic tries to consider both past and future by looking at tasks allocated in the past and those that are not allocated yet. In the first round the 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, in the second round it considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. In the second round, the algorithm performs more like the usual bin packing algorithms (i.e., tries to find the best bin for the current object), although it considers the remote blocking parameters while allocating a task to a processor. Any time the algorithm performs schedulability test, for more precise schedulability analysis, it always performs response time analysis [23].

The common steps of the algorithm before the two rounds are performed are as follow:

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

$$w_i = u_i + \lceil (\sum_{\rho_i < \rho_k} \mathrm{NC}_{i,k} \beta_{i,k} \lceil \frac{T_i}{T_k} \rceil + \mathrm{NC}_i \max_{\rho_i \geq \rho_k} \beta_{i,k}) / T_i \rceil \quad (1)$$

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

Considering the remote blocking terms of MPCP [6], the rationale behind the definition of weight is that the tasks that can be punished more by remote blocking become heavier. Thus, they can be allocated earlier and attract as many as possible of the tasks with which they share resources.

2. Macrotasks are generated, i.e., 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 in one processor, (i.e., it is not possible to schedule the macrotask on a single processor even if there is no any other tasks), it is set as broken, otherwise it is denoted as unbroken. Please observe that the test of fitting a macrotask in a single processor (to set it as broken or unbroken) is only done at the beginning. Later on at any time the algorithm tests fitting an unbroken macrotask in a processor, the macrotask may co-exist with other tasks and/or macrotasks on the same processor.

If a macrotask is unbroken, the partitioning algorithm always allocates 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 the utilizations (not weights) of its tasks . This is because all the tasks within an unbroken macrotask will always be allocated on the same processor and the tasks will not suffer from any remote blocking, hence there is no need to consider blocking parameters in the weight of an unbroken macrotask.

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 denote this list the *mixed list*.

The strategy of allocation of tasks in both rounds depends on attraction between tasks. The attraction function 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:

$$v_{i,k} = \begin{cases} \mathrm{NC}_{i,k} \beta_{i,k} \lceil \frac{T_i}{T_k} \rceil & \rho_i < \rho_k; \\ \mathrm{NC}_i \beta_{i,k} & \rho_i \geq \rho_k \end{cases} \quad (2)$$

The rationale of the attraction function is to allocate the tasks that may remotely block a task, τ_i, to the same processor as of τ_i (in order of the amount of remote blocking overhead) as far as possible. Please notice, the definition of weight (Equation 1) and attraction function (Equation 2) are heuristics that guide the algorithm under MPCP. However, these functions may differ under other synchronization protocols, e.g., MSRP and partitioned FMLP, which have different remote blocking terms.

There can be the case in which all tasks sharing resources end up in one macrotask. In this case if the macrotask can fit in one processor, there is no need to use MPCP or any other multiprocessor synchronization protocol, because there will not be any global resources in the system. On the other hand, if the macrotask does not fit in one processor (i.e., should be broken) the algorithm attempts, by using weight (Equation 1) and attraction (Equation 2) functions to put attracted tasks on the same processor as far as possible which leads to reducing the remote blocking overhead.

Now we present the continuation of the algorithm in two rounds:

First Round: After the common steps the following steps are repeated within the first round until all tasks are allocated to processors (partitions):
1. All processors are ordered in their non-increasing order of utilization.
2. The object at the top of the mixed list is picked. **(i)** If the object is a task, τ_i, and it does not belong to a broken macrotask (τ_i does not share any resource) τ_i will be allocated onto the first processor that fits it (all tasks on the processor are still schedulable), beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit τ_i a new processor is added to the list and τ_i is allocated onto it. **(ii)** If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits all of them. If none of the processors can fit the macrotask, it (all its tasks) will be allocated onto a new processor.**(iii)** If the object is a task, τ_i, 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 τ_i based on equation 2. We call this list the *attraction list* of τ_i. Task τ_i 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. As many as possible of the tasks from the attraction list are then allocated to the processor. 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., at least one of the processors become unschedulable, the first round fails and the algorithm moves to the second round and restarts.

Second Round: The following steps are repeated until all tasks are allocated to processors:
1. The object at the top of the mixed list is picked. **(i)** If the object is a task and it does not belong to a broken macrotask, this step is performed the same way as in the first round. **(ii)** If the object is an unbroken macrotask, in this the algorithm performs the same way as in the first round. **(iii)** If the object is a task, τ_i, that belongs to a broken macrotask, the processors are put in a ordered list, denoted as *Plist*. However the processors are put in *Plist* in two steps. First, the processors that include some tasks from τ_i's macrotask

are added to *Plist* in non-increasing order of processors' attraction to τ_i (according to equation 2), i.e., the processor which has the greatest sum of attractions of its tasks to the picked task (τ_i) is the most attracted processor to τ_i and is added to *Plist* first. Second, the processors that do not contain any task from τ_i's macrotask are added to *Plist* in non-increasing order of their utilization. After the two steps, the processors which contain at least one task from τ_i's macrotask will be located at the top of the ordered list, *Plist*, followed by the processors not containing any task from τ_i's macro task. The rationale behind this is that the algorithm first attempts to allocate τ_i on a processor containing some tasks from τ_i's macro task and if not succeeded then it tries other processors. The picked task (τ_i) will be allocated onto the first processor from the processor list (*Plist*) that will fit τ_i. Task τ_i will be allocated to a new processor if none of the existing ones can fit it. And the second round of the algorithm fails if allocating the task to the new processor makes some of the processors unschedulable.

If both rounds fail to schedule a task set the algorithm fails. If one of the rounds fails the result will be the output of the other one. If both rounds succeed to schedule the task set, the one with fewer partitions (processors) will be the output of the algorithm.

3.2 Synchronization-Aware Partitioning Algorithm (SPA)

We have implemented the best known existing partitioning algorithm proposed in [5] in our experimental evaluation framework. The implementation of the algorithm required details of the algorithm which were not presented in [5], hence, in this section we present the algorithm in more details.

1. First, the macrotasks are generated. In [5], macrotasks are denoted as bundles. A number of processors (enough processors that fit the total utilization of the task set) are added.

2. The macrotasks together with other tasks are ordered in a list in non-increasing order of their utilization. The algorithm attempts to allocate each macrotask (i.e., allocate all tasks within the macrotask) onto a processor. Without adding any new processor, all macrotasks and tasks that fit are allocated onto the processors. The macrotasks that can not fit are put aside. After any allocation, the processors are ordered in their non-increasing order of utilization.

3. The remaining macrotasks are ordered in the order of the cost of breaking them. The cost of breaking a macrotask is defined based on the estimated cost (blocking overhead) introduced into the tasks by transforming a local resource into a global resource (i.e., the tasks sharing the resource are allocated to different processors). The estimated cost of transforming a local resource R_q into a global resource is calculated as follows:

$$\mathrm{Cost}(R_q) = \text{Global Overhead} - \text{Local Discount} \tag{3}$$

The Global Overhead is calculated as follows:

$$\text{Global Overhead} = \max(|Cs_q|) / \min_{\forall \tau_i}\{\rho_i\} \tag{4}$$

where $\max(|Cs_q|)$ is the length of longest critical section accessing R_q.

The Local Discount is defined as follows:

$$\text{Local Discount} = \max_{\forall \tau_i \text{ accessing } R_q} (\max(|Cs_{i,q}|)/\rho_i) \quad (5)$$

where $\max(|Cs_{i,q}|)$ is the length of longest critical section of τ_i accessing R_q.

The cost of breaking any macrotask, mTask_k, is calculated as the summation of blocking overhead caused by transforming its accessed resources into global resources.

$$\text{Cost}(\text{mTask}_k) = \sum_{\forall R_q \text{ accessed by } \text{mTask}_k} \text{Cost}(R_q) \quad (6)$$

4. The macrotask with minimum breaking cost is picked and is broken in two pieces such that the size of one piece is as close as the largest utilization available among processors. This means, tasks within the selected macrotask are ordered in decreasing order of their size (utilization) and the tasks from the ordered list are added to the processor with the largest available utilization as far as possible. In this way, the macrotask has been broken in two pieces; (i) the one including the tasks allocated to the processor and (ii) the tasks that could not fit in the processor. If the fitting is not possible a new processor is added and the whole algorithm is repeated again.

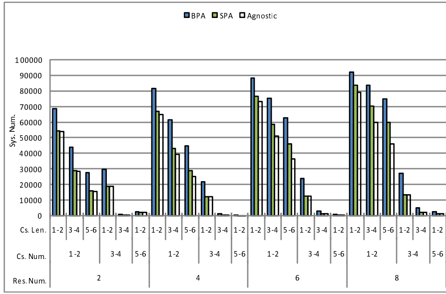
Firstly, as one can see, the SPA algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its utilization, i.e. the tasks are ordered in order of their utilization only. However, our algorithm assigns a weight (Equation 1) which besides the utilization includes the blocking terms as well. Secondly, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle (macrotask) is considered which could help to allocate tasks from a broken bundle to appropriate processors to decreases the blocking times. In our heuristic, we have defined an attraction function (Equation 2), which attempts to allocate the most attracted tasks from the current task's broken macrotask, on a processor. As the experimental evaluation in Section 4 shows, considering these issues can improve the partitioning significantly.

4 Experimental Evaluation and Comparison of Algorithms

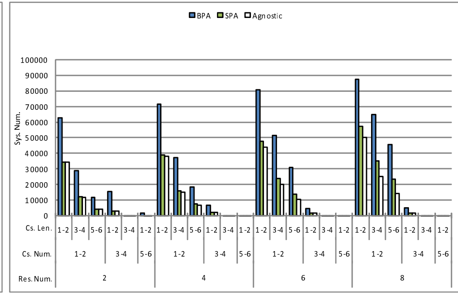
In this section we present our experimental results of our blocking-aware bin-packing algorithm (BPA) together with the blocking-aware algorithm recently proposed in [5] (SPA), as well as the reference blocking-agnostic algorithm. For a number of systems (task sets), we have compared the performance of the algorithms in two different aspects; (1) Given a number of systems, the total number of systems that each of the algorithms can schedule, (2) The processor reduction aspect of algorithms.

4.1 Experiment Setup

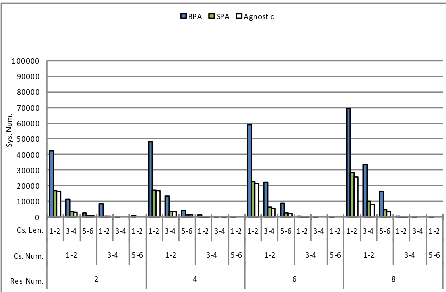
We generated systems (task sets) for different workloads; we denote workload as a defined number of fully utilized processors, e.g., the workload equal to 3 fully utilized



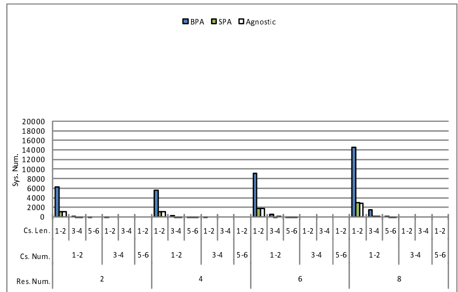
(a) Workload: 3 processors, 3 tasks per processor



(b) Workload: 3 processors, 6 tasks per processor



(c) Workload: 3 processors, 9 tasks per processor



(d) Workload: 6 processors, 6 tasks per processor

Fig. 1. Total number of task sets each algorithm schedules

processors means the summation of utilizations of all tasks in the system equals to 3. Please notice that the definition of the workload as a number of processors is only to show the total utilization of the task set and it is not the same as the number of required processors (which may be more than the workload) to schedule the task set. Given a workload, the full capacity of each processor (utilization of 1) is randomly divided among a defined number of tasks. 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 6 and the maximum length of any critical section to 6 time units, hence the WCET of each task is greater than 36 (6×6) time units. The WCET of each task was randomly chosen between 36 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, 6 or 9 tasks per processor), e.g., 3 tasks per processor means that the utilization of one processor (utilization $= 1$) is randomly distributed among 3 tasks.

3. The number of resources (2, 4, 6, or 8). For each alternative, the resource accessed by each critical section is randomly chosen among the resources, e.g, given the alternative with 2 resources (R_1 and R_2), the resource accessed by any critical section is randomly chosen from $\{R_1, R_2\}$.
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). For an alternative (e.g., 1 to 2 critical sections per task), the number of critical sections of any task τ_i is randomly chosen from $\{1, 2\}$.
5. The range of length of critical sections (1 to 2, 3 to 4, or 5 to 6). The length of each critical section is chosen the same way as the number of critical sections per task.

For each setting, we generated 100.000 systems, and combining the parameters of settings, i.e., (workloads)×(tasks per processor)×(resources)×(critical sections per task)×(critical section lengths)= $4 \times 3 \times 4 \times 3 \times 3 = 432$ different settings, total number of systems generated for the experiment were 43.200.000.

With the generated systems we were able to evaluate the partitioning algorithms 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.

4.2 Results

In this section we present the evaluation results of our proposed blocking-aware algorithm (BPA), an existing blocking-aware algorithm [5] (SPA) and the blocking-agnostic algorithm.

The first aspect of comparison of the results from the algorithms is, given a number of systems, the total number of systems each algorithm successfully schedules (Figure 1). Figures 1(a), 1(b) and 1(c) represent the results for 3, 6 and 9 tasks per processor 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 per each task are between 1 and 2, and the length of these critical sections are between 1 and 2 time units. For some settings the number of schedulable systems were too few to be shown on the graphs, thus we omitted these settings from the graphs, e.g., The results for the combination of the number of critical sections = 3-4 and the length of critical sections = 5-6 are not shown in Figure 1.

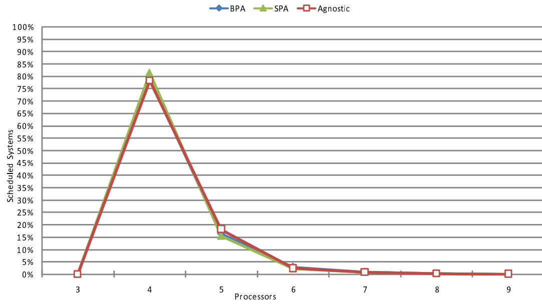
As depicted in Figure 1, considering the total number of systems that each algorithm succeeds to schedule, our blocking-aware algorithm (BPA) performs better (can schedule more systems) compared to the SPA and the blocking-agnostic algorithm. However the SPA performs better than the blocking-agnostic algorithm. As shown in the figure, by increasing the number of resources, the number of successfully scheduled systems in all algorithms is 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 illustrated that the blocking-aware algorithms perform better as the number of resources is increased. It is also shown that increasing the number and/or the length of critical sections generally reduces the number of schedulable systems significantly. The reason is that more and longer critical sections introduce greater blocking overhead into the tasks making fewer systems schedulable.

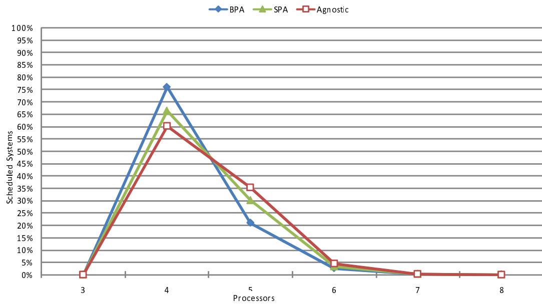
As the number of tasks per processor is increased from 3 (Figures 1(a)) to 6 (Figures 1(b)) and to 9 (Figures 1(c)), the BPA performs significantly better (i.e., schedules significantly more systems) than the SPA and blocking-agnostic bin-packing. However, as one can see, the SPA does not perform significantly better than the blocking-agnostic algorithm as the number of tasks per processor are increased. Increasing the number of tasks per processor lead to smaller tasks (tasks with smaller u_i). The BPA allocates tasks from a broken macrotask based on Equations 1 and 2, which are functions of the blocking parameters (the number and length of critical sections) as well as the size of the tasks. On the other hand, with the smaller size of tasks, the blocking parameters have a bigger role in these functions, hence more dependent tasks are allocated to the same processor. This leads to less blocking overhead and increased schedulability, hence more systems are scheduled by BPA as the tasks per processor are increased. On the other hand, in SPA, allocation of tasks from a broken macrotask is only based on their utilization, and this does not necessarily allocates highly dependent tasks to the same processor.

As the workload (the number of fully utilized processors) is increased, although the BPA still performs better than the SPA and the blocking-agnostic algorithm, generally the number of schedulable systems by all algorithms is significantly reduced (Figure 1(d)). The reason for this behavior is that the number of tasks within systems are relatively many (36 tasks per each system in Figure 1(d)) and the workload is high (6 fully utilized processors), and all the tasks within systems share resources. On the other hand, the MPCP is pessimistic. 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 benefit from our partitioning algorithm to increase the performance.

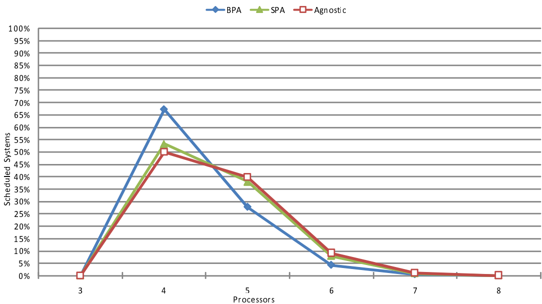
The second aspect for comparison of performance of the algorithms is the processor reduction aspect. To show this, for each algorithm, we ordered the total schedulable systems in order of the number of required processors. Figure 2 illustrates the results for the workload of 3 fully packed processors and different number of tasks (3, 6 and 9) per processor. For each algorithm, the schedulable systems by each number of processors are shown as percentage of the total scheduled systems by that algorithm. As the results show, for 3 tasks per processor all three algorithms perform almost the same (Figure 2(a)), i.e., each algorithm schedules around 80% of its schedulable systems by 4 processors, 15% to 18% by 5 processors and less than 3% by 6 processors, etc. The reason is that the tasks are large (the utilization of a processor is distributed among 3 task), thus the blocking-aware algorithms do not have much possibility to increase the



(a) 3 tasks per processor



(b) 6 tasks per processor



(c) 9 tasks per processor

Fig. 2. Percentage of systems each algorithm schedules, ordered by required number of processors

performance. However as the number of tasks per processor is increased (Figures 2(b) and 2(c) for 6 and 9 tasks per processor respectively), the blocking-aware algorithms, generally, perform better in processor reduction aspect. Especially the BPA, performs significantly better than the the SPA and the blocking-agnostic algorithm. This means that BPA reduces the required number of processors compared to SPA and the blocking-agnostic algorithm, e.g., as shown in Figure 2(c), 68% and 28% of the systems scheduled by BPA require 4 and 5 processors respectively, while 54% and 37% of systems

scheduled by SPA can be scheduled by 4 and 5 processors respectively. This means a bigger part (68%) of systems scheduled by BPA require only 4 processors while with SPA this number is smaller (54%).

5 Conclusion

In this paper we have proposed a heuristic blocking-aware algorithm, for identical unit-capacity 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 schedulability of a task set and may lead to fewer required processors compared to blocking-agnostic bin-packing algorithms. We have also presented and implemented an existing similar blocking-aware algorithm originally proposed in [5].

Since in practice most systems use fixed priority scheduling protocols, we have developed our algorithm under MPCP, a standard synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling. Another reason to implement our algorithm under MPCP was to be able to compare our approach to the existing similar approach [5] which has also been developed under MPCP. However, our approach is not limited to MPCP and it can easily be extended to other synchronization protocols such as MSRP and partitioned FMLP.

Our experimental results confirm that our algorithm mostly performs significantly better than the blocking-agnostic as well as the existing heuristic with respect to the number of schedulable systems and the number of required processors. However, given a NP-hard problem, a bin-packing algorithm may not achieve the optimal solution, i.e, there can exist systems that only one of the algorithms can schedule. Thus using a combination of heuristics 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. A very interesting future work is to apply our approach to different synchronization protocols and investigate the effect of bin-packing on those protocols and compare the improvement in their performance. 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.

Acknowledgments

The authors wish to thank Karthik Lakshmanan for fruitful discussions, helping out in improving the quality of this paper.

References

1. Baker, T.: A comparison of global and partitioned EDF schedulability test for multiprocessors. Technical report (2005)
2. Baker, T.: Stack-based scheduling of real-time processes. Journal of Real-Time Systems 3(1), 67–99 (1991)
3. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S.: A categorization of real-time multiprocessor scheduling problems and algorithms. In: Handbook on Scheduling Algorithms, Methods, and Models. Chapman Hall/CRC, Boca (2004)
4. Devi, U.: Soft real-time scheduling on multiprocessors. In: PhD thesis (2006), http://www.cs.unc.edu/~anderson/diss/devidiss.pdf
5. Lakshmanan, K., de Niz, D., Rajkumar, R.: Coordinated task scheduling, allocation and synchronization on multiprocessors. In: Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS 2009), pp. 469–478 (2009)
6. Nemati, F., Nolte, T., Behnam, M.: Blocking-aware partitioning for multiprocessors. Technical report, Mälardalen Real-Time research Centre (MRTC), Mälardalen University (March 2010), http://www.mrtc.mdh.se/publications/2137.pdf
7. Rajkumar, R.: Synchronization in Real-Time Systems: A Priority Inheritance Approach. Kluwer Academic Publishers, Dordrecht (1991)
8. Tindell, K.W., Burns, A., Wellings, A.J.: Allocating hard real-time tasks: An NP-hard problem made easy. Journal of Real-Time Systems 4(2), 145–165 (1992)
9. Altenbernd, P., Hansson, H.: The slack method: A new method for static allocation of hard real-time tasks. Journal of Real-Time Systems 15(2), 103–130 (1998)
10. de Niz, D., Rajkumar, R.: Partitioning bin-packing algorithms for distributed real-time systems. Journal of Embedded Systems 2(3-4), 196–208 (2006)
11. Baruah, S., Fisher, N.: The partitioned multiprocessor scheduling of sporadic task systems. In: Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS 2005), pp. 321–329 (2005)
12. Rajkumar, R., Sha, L., Lehoczky, J.P.: Real-time synchronization protocols for multiprocessors. In: Proceedings of the 9th Real-Time Systems Symposium, RTSS 1988 (1988)
13. Gai, P., Lipari, G., Natale, M.D.: 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 2001), pp. 73–83 (2001)
14. Gai, P., Di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., Marceca, P.: 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 2003), pp. 189–198 (2003)
15. López, J.M., Díaz, J.L., García, D.F.: Utilization bounds for EDF scheduling on real-time multiprocessor systems. Journal of Real-Time Systems 28(1), 39–68 (2004)
16. Devi, U., Leontyev, H., Anderson, J.: Efficient synchronization under global EDF scheduling on multiprocessors. In: Proceedings of 18th IEEE Euromicro Conference on Real-time Systems (ECRTS 2006), pp. 75–84 (2006)
17. Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: A flexible real-time locking protocol for multiprocessors. In: Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), pp. 47–56 (2007)
18. Brandenburg, B., Calandrino, J., Block, A., Leontyev, H., Anderson, J.: 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 2008), pp. 342–353 (2008)

19. Brandenburg, B., Anderson, J.: An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In: Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008), pp. 185–194 (2008)
20. Brandenburg, B.B., Anderson, J.H.: A comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 105–124. Springer, Heidelberg (2008)
21. Easwaran, A., Andersson, B.: Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In: Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS 2009), pp. 377–386 (2009)
22. Nemati, F., Behnam, M., Nolte, T.: Efficiently migrating real-time systems to multi-cores. In: Proceedings of 14th IEEE Conference on Emerging Techonologies and Factory, ETFA 2009 (2009)
23. Burns, A.: Preemptive priority based scheduling: An appropriate engineering approach. In: Principles of Real-Time Systems, pp. 225–248. Prentice Hall, Englewood Cliffs (1994)