Efficient Partitioning of Sporadic Real-Time Tasks with Shared Resources and Spin Locks

Alexander Wieder Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—Partitioned fixed-priority scheduling is widely used in embedded multiprocessor real-time systems due to its simplicity and low runtime overheads. However, it fundamentally requires a static mapping of tasks to processors to be determined. Optimal task set partitioning is known to be NP-hard, and the situation is further aggravated when limited resources (such as I/O ports, co-processors, buffers, *etc.*) must be shared among the tasks. Partitioning heuristics are much faster to compute, but may fail to find a valid mapping even if one exists. In practice, such inefficiencies can be addressed by over-provisioning processors (*i.e.*, by using more and faster processors than strictly required), albeit at the expense of increased space, weight, and power (SWaP) requirements.**

This work makes two contributions towards the efficient mapping of real-time tasks that share resources protected by spin locks. First, an Integer Linear Programming (ILP) formulation of the problem is presented, which, while computationally expensive, is efficient in the sense that it will find a valid assignment if one exists, thereby minimizing processor requirements. This ILP formulation is the first optimal solution to the mapping problem in the presence of spin locks. Second, a new resource-aware partitioning heuristic is introduced, which, while not optimal, is efficient in the sense that it easily scales to large problem instances. Notably, the proposed heuristic is much simpler than prior approaches, parameter-free, and shown to perform well for a wide range of workloads.

I. Introduction

Partitioned fixed-priority (P-FP) scheduling, under which tasks are statically distributed across all processors and each processor is individually scheduled by a local fixed-priority scheduler, is the *de facto* standard in embedded multiprocessor real-time systems today. For instance, P-FP scheduling is mandated by the widely adopted AUTOSAR standard for automotive systems [1], and also supported by virtually all POSIX-compliant real-time OSs such as VxWorks, QNX, LynxOS, *etc.*

P-FP scheduling is attractive because it is well understood and trivial to implement with low runtime overheads; however, it also requires a valid *partitioning* to be provided, that is, a static mapping of tasks to processors under which all tasks are *schedulable* (*i.e.*, guaranteed to satisfy all timing requirements). Computing such a partitioning can be challenging. In fact, even if tasks are *independent*, that is, if they do not share any resources besides the processor, finding a valid assignment requires solving a bin-packing-like problem [14], which is known to be NP-hard, and for which heuristics are typically used (*e.g.*, see [8, 14, 17]).

In practice, tasks often share limited resources such as I/O devices, co-processors, or message buffers, which requires the use of *locks* to serialize conflicting accesses. For example, the AUTOSAR specification [1] mandates the availability of *spin locks* for this purpose. Importantly, when using locks, tasks are no longer fully independent as they may be subject to *blocking*, which must be bounded and accounted for during schedulability analysis. Crucially, as we review in Sec. III-B, *the extent to which tasks are subject to blocking depends on how resource-sharing tasks are assigned to processors*. Classic bin-packing heuristics, however, do not consider blocking and thus are liable to expose tasks to excessive blocking.

Such inadvertently wasteful task allocation can prevent multicore platforms from being utilized efficiently, in the sense that additional or faster processors may appear necessary, even though, given a better partitioning, a simpler and cheaper platform could have sufficed (we provide empirical evidence supporting this observation in Sec. IV). In many embedded application domains such as avionics, automotive systems, or mobile systems, such a superfluous increase in space, weight, and power (SWaP) requirements—and ultimately cost—can render a product technologically or economically unviable. To realize the full potential of embedded multicore platforms, *resource-aware* partitioning approaches are needed.

In this work, we approach the problem of partitioning a set of tasks that share resources protected by spin locks in two ways, reflecting two interpretations of "efficient." First, in Sec. V, we present an *exact* approach that uses *integer linear programming* (ILP) to assign each task a processor and a unique priority such that all tasks are schedulable under the MSRP [18], a real-time locking protocol for shared-memory systems based on spin locks (reviewed in Sec. III-B). Our ILP-based approach is optimal in that it yields a valid partitioning and priority assignment if one exists (with regard to the underlying schedulability analysis of the MSRP [18]). To the best of our knowledge, this is the first partitioning method that is optimal in the presence of shared resources protected by spin locks, which is made possible by the novel approach of encoding the MSRP analysis [18] as ILP constraints. Our ILP-based approach is efficient in the sense that it prevents over-provisioning, but due to the NP-hardness of the assignment problem, it also faces inherent scalability limits, which we empirically evaluate in Sec. VII-A.

Second, in Sec. VI, we present a novel resource-aware partitioning *heuristic*, called *Greedy Slacker*, that often produces valid partitionings and priority assignments even in cases where prior heuristics fail. Greedy Slacker greedily assigns tasks such that the least slack (*i.e.*, the difference between maximum response time and deadline) among all tasks is maximized. This approach is much simpler than prior resource-aware heuristics (see Sec. II), yet our evaluation in Sec. VII-B shows that it delivers equal-or-better schedulability in a wide range of

scenarios. While Greedy Slacker is not optimal, it is efficient in the sense that it scales to large problem instances.

In this paper, we focus on spin locks (where blocked tasks busy-wait) rather than on *semaphores* (where blocked tasks suspend) due to the considerable practical relevance of spin locks in the context of AUTOSAR [1], and because the partitioning problem in the presence of spin locks, despite their widespread use, has not been studied in prior work, which we review next.

II. Related Work

Allocation problems similar to bin-packing [19] arise in a vast range of settings; due to space constraints, we restrict our focus to prior works most relevant to the partitioning of real-time workloads onto multiprocessor platforms.

The partitioned scheduling of independent sporadic tasks is by now a well-understood problem [6, 13]; Fisher provides a comprehensive discussion [16]. To cope with the inherent complexity of exact partitioning approaches, bin-packing heuristics are commonly applied (*e.g.*, see [8, 14, 16, 17]). In recent work on near-optimal partitioning of independent tasks, Baruah presented a polynomial-time approximation scheme [6] and Chattopadhyay and Baruah showed how to leverage lookup tables to enable fast, yet arbitrarily accurate partitioning [12].

As discussed in Sec. I, for task sets with shared resources, generic bin-packing heuristics can be inefficient since they do not take blocking due to resource requests into account. To consider this additional blocking, Lakshmanan *et al*. presented a partitioning heuristic tailored to the MPCP, a semaphore-based multiprocessor real-time locking protocol [20]. This heuristic organizes tasks sharing resources into groups in order to assign them to the same processor. In subsequent work, Nemati *et al*. presented BPA [23], another partitioning heuristic for the MPCP following the same approach, which employs advanced cost heuristics to more accurately identify group splits with low overall blocking. These resource-aware partitioning heuristics, which we discuss in more detail in Sec. III-C, are tailored to produce a valid partitioning where generic bin-packing heuristics fail. However, these resource-aware heuristics have not yet been studied in the context of spin locks and are not directly applicable (due to their protocol-specific nature). They also do not necessarily always find a valid partitioning if one exists.

In contrast, *exact* partitioning approaches are optimal in that they fail to produce a valid partitioning only if no such partitioning exists. Exact ILP-based approaches for the partitioning of *independent* tasks under EDF and FP scheduling were presented by Baruah [9] and Baruah and Bini [7]. Targeting *boolean satisfiability* (SAT) instead of linear programming as the underlying formalism, Metzner and Herde proposed RTSAT [22], which first transforms an ILP-formulation (similar to ours) to a SAT instance, and then employs a specialized SAT solver. For task sets with precedence constraints, Zheng *et al*. [28] presented an ILP formulation that explicitly considers interference due to communication on a shared bus. Most closely related to our work is an ILP formulation by Zeng and Di Natale [27], who recently incorporated blocking due to *local* resource sharing (*i.e.*, due to resources accessed only on one processor).

In the context of multicore platforms, a common limitation of all of the above-cited ILP formulations is that they either do not consider shared resources at all, or only local shared resources. In contrast, our objective is the efficient partitioning of task sets with *global* shared resources (*i.e.*, resources accessed on multiple processors) that are protected by spin locks.

III. System Model and Background

We begin by introducing the assumed task and system model, review the MSRP, and summarize prior partitioning heuristics.

A. System Model

1) Task Model: We consider a task set τ consisting of n tasks $T_1, \ldots, T_n$ that each *sporadically* release a sequence of jobs. The minimum separation of two consecutive job releases of a task T_i is denoted by p_i, and the maximum execution requirement of each job is denoted by e_i. Once the j^{th} job of T_i arrives at time $a_{i,j}$, its execution requirement must be fulfilled within d_i time units, where d_i denotes T_i's *relative deadline*. Deadlines are *constrained*, that is, $d_i \leq p_i$ for each task T_i. In our model, we allow jobs to incur *release jitter* to model precedence constraints for tasks [26]. That is, after arrival of T_i's v^{th} job at time $a_{i,v}$, it may take up to j_i time units before the job is *released* and becomes available for execution.

2) Scheduling: We consider the use of partitioned task-level fixed-priority scheduling, which is commonly used for embedded industrial applications (*e.g.*, it is mandated by AUTOSAR [1]). Each task T_i is statically assigned a priority $1 \leq \pi_i \leq n$ that does not change during runtime and is also used by all jobs released by T_i. Here, *lower* numerical values of π_i correspond to $higher$ scheduling priorities. Further, each task T_i is statically assigned to exactly one processor, denoted as $P(T_i)$, on which all jobs of T_i are executed. On each processor, among all *runnable* jobs, *i.e.*, jobs that are released but not yet complete and not suspended, the job with highest (*i.e.*, numerically lowest) priority is executed. There are m identical processors in total.

3) Shared Resources: Each job of T_i in τ may issue requests for one or more shared resources. We let n_r denote the total number of shared resources in the system, and let l_q with $1 \leq q \leq n_r$ denote the q^{th} resource. For each l_q, we denote the maximum number of times a job of T_i may access l_q with $N_{i,q}$, and let $L_{i,q}$ denote the maximum critical section length of any such request (where $L_{i,q} = 0$ if $N_{i,q} = 0$).

4) Overheads: In a real system, tasks are subject to overheads such as context switch costs or the loss of cache affinity when preempted. We assume that all non-negligible overheads have already been factored into the relevant task parameters (*i.e.*, mainly e_i and each $L_{i,q}$) using standard accounting techniques (see [10, Chs. 3 and 7] for a detailed discussion). Next, we review the real-time locking protocol that mediates resource access.

B. The Multiprocessor Stack Resource Policy

The *Multiprocessor Stack Resource Policy* (MSRP) [18] is a shared-memory locking protocol that enables predictable access to shared resources. The MSRP distinguishes *global* and *local* resources: *global resources* are accessed from multiple processors, whereas all tasks accessing a *local resource* must

reside on the same processor. For local resources, the MSRP uses the classic uniprocessor *Stack Resource Policy* (SRP) [5].

The SRP, sometimes also called the *immediate priority ceiling protocol*, is based on *resource ceilings* that are defined for each resource l_q as the highest priority of any task accessing l_q: $\Pi(l_q) = \min_{T_i}\{\pi_i | N_{i,q} > 0\}$. Further, a dynamic *system ceiling* $\hat{\Pi}(t)$ is defined as the highest resource ceiling of any resource l_q in use at time t: $\hat{\Pi}(t) = \min_{l_q}\big\{\{\Pi(l_q) | l_q \text{ locked at time } t\} \cup \{n+1\}\big\}$. The key scheduling rule of the SRP is that a newly released job of T_i may only start executing at time t if $\pi_i < \hat{\Pi}(t)$, which implies that all required resources are available.

For global resources, *i.e.*, resources accessed from different processors, the MSRP cannot use the uniprocessor SRP, which relies on per-processor ceilings and does not generalize to multiprocessor systems. Instead, the MSRP uses non-preemptive FIFO spin locks to coordinate access to global resources: to gain access to a global resource l_q, a job becomes non-preemptive and starts spinning until it gains access to l_q. Concurrent requests by jobs on other processors to the same resource are served in FIFO order. Once a job finishes its critical section, it becomes preemptive again and normal scheduling resumes.

Non-preemptive spin locks are in widespread use. For instance, the AUTOSAR 4.0 specification provides the `SuspendAllInterrupts()` and `GetSpinlock()` APIs to initiate non-preemptive execution and to acquire spin locks, respectively. The MSRP can thus be realized in today's systems.

Concurrent access to resources leads to *blocking*. Next, we review Gai *et al.*'s analysis of blocking under the MSRP [18].

1) Blocking under the MSRP: Under the MSRP, jobs are subject to three types of blocking: *local blocking* and *non-preemptive blocking*, which cause priority inversions [11, 25], and *remote blocking*, which results in spinning.

a) Local Blocking: A job of T_i may incur local blocking if a job of a local lower-priority task T_j executes a request for a local resource l_q with $\Pi(l_q) \leq \pi_i$. Under the SRP, T_j's request for l_q causes the system ceiling $\hat{\Pi}(t)$ to be set to *at least* $\Pi(l_q)$. If T_i releases a job while T_j is holding l_q, T_i suspends since $\hat{\Pi}(t) \leq \pi_i$, and hence T_i's job is blocked by T_j's job. Each job of T_i can be locally blocked at most once (upon release) for a duration of at most β_i^{loc} time units, where

$$\beta_i^{loc} = \max_{T_j,q}\{L_{j,q} | N_{j,q} > 0 \wedge \Pi(l_q) \leq \pi_i < \pi_j \wedge l_q \text{ local}\}.$$

b) Remote Blocking: When using non-preemptive FIFO spin locks, a job of T_i spins non-preemptively until it acquires a requested global resource l_q. This spin time is bounded by the sum of maximum critical section lengths for l_q from each other processor, denoted as $S_{i,q}$, where $S_{i,q} = 0$ if $N_{i,q} = 0$, and

$$S_{i,q} = \sum_{1 \leq k \leq m, k \neq P(T_i)} \max\{L_{x,q} | P(T_x) = k\} \quad \text{if } N_{i,q} > 0.$$

We let β_i^{rem} denote an upper bound on the total remote blocking incurred by any job of T_i, where $\beta_i^{rem} = \sum_q N_{i,q} \cdot S_{i,q}$.

c) Non-Preemptive Blocking: A lower-priority job of T_j spinning or executing non-preemptively can cause a job of T_i to incur a priority inversion. The duration β_i^{NP} of this non-preemptive blocking is bounded by T_j's worst-case spin time and critical section length for a single request:

$$\beta_i^{NP} = \max_{T_j,q}\big\{S_{j,q} + L_{j,q} | P(T_i) = P(T_j) \wedge \pi_i < \pi_j \wedge l_q \text{ is global}\big\}.$$

2) Schedulability Analysis: Response-time analysis [3] is used to determine if a task T_i is schedulable. Under the MSRP, T_i's *response time* R_i satisfies the following recursion [18]:

$$R_i = e_i + \beta_i^{rem} + \max\big\{\beta_i^{NP}, \beta_i^{loc}\big\} + \sum_{T_h, \pi_h < \pi_i \wedge P(T_i) = P(T_h)} \left\lceil \frac{R_i + j_h}{p_h} \right\rceil \cdot (e_h + \beta_h^{rem}).$$

Given the response time R_i of a task T_i, its schedulability can be determined by checking whether the condition $R_i + j_i \leq d_i$ is satisfied. Note that the response time of a task crucially depends on the assignment of tasks to processors and its priority. For the scope of this work, the priority assignment and the mapping of tasks to processors is not assumed to be given and fixed, but rather yet to be determined. Next, we review *partitioning heuristics* commonly used to assign tasks to processors.

C. Partitioning Heuristics

Generic bin-packing heuristics are commonly used to map tasks to processors. Bin-packing heuristics distribute a set of different objects (tasks) of a given size (processor utilization) to bins (processors), such that each object is assigned to exactly one bin and the total size of all objects assigned to a bin does not exceed the bin's capacity (all tasks are schedulable).

1) Classic Bin-Packing Heuristics: Commonly used heuristics include the *first-fit*, *next-fit*, *best-fit* and *worst-fit* heuristics [19], which we describe in brief. All heuristics take a sequence of objects as input and successively assign them to bins. The *first-fit* heuristic iterates over all bins in the order they were allocated, and assigns the current object to the first bin with sufficient remaining capacity. If no such bin exists, it allocates a new bin and assigns the current object to it. The *next-fit* is simpler in that it only checks the last allocated bin and allocates a new bin if the last allocated bin does not have sufficient capacity to fit the current object. Both the *first-fit* and *next-fit* heuristics report a failure if the maximum number of bins is exceeded. The *best-fit* and *worst-fit* heuristics allocate the maximum number of bins upfront and then assign each object to a bin such that the remaining capacity in that bin is minimized or maximized, respectively. If no bin with sufficient capacity exists, they report a failure. The *any-fit* heuristic, which we denote as AF in the following, subsumes all previously described bin-packing heuristics in that it tries all of them (in the order *worst-fit*, *best-fit*, *first-fit*, *next-fit*) and returns the first successfully computed result. For all heuristics, we consider the input being processed in order of decreasing size, which typically results in a lower number of required bins [19].

2) Resource-Aware Partitioning Heuristics: Resource sharing causes dependencies among tasks (*i.e.*, β_i^{loc}, β_i^{rem} and β_i^{NP}) that are not considered by generic bin-packing heuristics. Resource-aware partitioning heuristics account for these effects and take the resource access patterns into account when mapping tasks to processors. We outline the *MPCP partitioning heuristic* [20] and the *Blocking-Aware Partitioning Algorithm (BPA)* [23].

Lakshmanan *et al*. proposed the MPCP partitioning heuristic for the *multiprocessor priority ceiling protocol* (MPCP) [24], a locking protocol for shared-memory multiprocessor systems. Under the MPCP partitioning heuristic, tasks are assigned to the same *bundle* if they share (possibly transitively) a common set of resources. Bundles are then assigned to processors using the best-fit heuristic. This leads to tasks accessing the same resources being assigned to the same processor, if possible, to avoid the need for inter-processor synchronization. Bundles that do not fit on any processor are *broken* into multiple smaller ones, such that one bundle fits as tightly as possible onto the processor with the highest remaining capacity. Bundles are assigned and broken (if necessary) until all tasks are assigned.

The BPA presented by Nemati *et al*. [23] is related to the MPCP partitioning heuristic in that it groups together tasks that access the same resource(s), and, if possible, assigns all tasks belonging to a group to the same processor. Otherwise, task groups are split and the respective tasks are assigned to different processors. In this case, for each pair of tasks, the BPA also takes into account the remote blocking (estimated based on the resource access patterns of each task) that can be caused when assigning two tasks to different processors. Since the BPA is tailored to the MPCP, we adjusted the blocking estimation functions specific to the MPCP to instead reflect the blocking as incurred under the MSRP.

Although resource-aware heuristics can perform better than resource-oblivious heuristics (*e.g.*, see the experiments in [20, 23]), it is still unclear how they compare with exact approaches. In particular, do resource-aware heuristics in practice already yield results close to optimal, or is there still a significant margin for improvement? As shown next, even resource-aware heuristics may leave a considerable potential wasted.

IV. The Case for Optimal Partitioning

Exact partitioning approaches for task sets with shared resources can be computationally expensive due to the hardness of the underlying bin-packing problem. This complexity raises the question whether exact approaches can offer substantial benefit over resource-aware heuristics. To answer this question, we conducted an experiment to evaluate whether there exists a potential that is left unused by heuristics but could be exploited by an exact approach. To this end, we generated task sets for which a valid partitioning was known to exist by construction, and hence an exact partitioning approach would have found a valid partitioning. Then we let resource-oblivious and resource-aware heuristics partition the same task sets and checked schedulability of the computed partitionings. Priorities were assigned in a rate-monotonic fashion [21], and before assigning a task to a processor (*i.e.*, to determine whether a task "fits") a response-time schedulability test was applied to rule out choices that render the task set unschedulable.

Fig. 1 shows the fraction of schedulable task sets under each partitioning heuristic depending on the number of tasks in the system. The straight line at the top of the graph marks the fraction of task sets that *can* be successfully partitioned by an exact approach, that is, all task sets as only partitionable task sets were considered in this experiment. As is apparent from

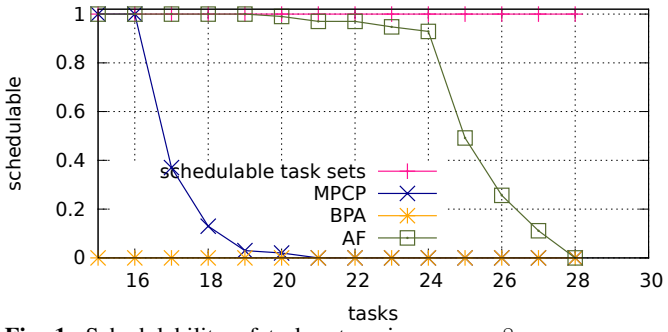


Fig. 1: Schedulability of task sets using $m = 8$ processors and 16 resources. Critical section lengths were randomly chosen from $[1us, 100us]$, task periods were randomly chosen from $[3ms, 33ms]$, and the average utilization per task was set to 0.1. See Sec. VII-B for details on the task set generation procedure.

Fig. 1, AF is able to produce valid partitionings for all task sets consisting of up to 20 tasks. For larger task sets, AF is unable to produce valid partitionings for a large fraction of the generated task sets although valid partitionings exist and hence, an optimal partitioning scheme would have found them. Both the MPCP heuristic and BPA show surprisingly low schedulability, an effect we revisit in Sec. VII-B.

While Fig. 1 shows results for specific parameter choices, similar results can be obtained for many other configurations: if blocking due to resource sharing constitutes a "bottleneck" with respect to schedulability, then an ill-chosen task assignment can render a partitioning invalid. Clearly, for task sets in which blocking durations are not significant, resource-oblivious heuristics may yield results comparable to resource-aware heuristics. However, as demonstrated in Fig. 1, if blocking is not negligible, then there exists a significant potential to be exploited by an exact approach. Next, we present such an approach based on a novel ILP encoding of the partitioning problem.

V. ILP-Based Processor and Priority Assignment

In this section, we present our ILP formulation of the task set partitioning and priority assignment problem in the presence of MSRP-arbitrated shared resources. This approach is optimal with regard to the MSRP analysis, that is, if a solution under the MSRP analysis exists, a valid partitioning and priority assignment will be found. Initially, the ILP formulation does not specify an objective function. That is, we accept any solution that satisfies all constraints of the ILP, which allows the objective function to be used to optimize other criteria (such as the required number of processors, see Sec. V-D below).

We consider the jitter j_i, the deadline d_i, the cost e_i, the period p_i, the maximum request length $L_{i,q}$, and the maximum number of requests $N_{i,q}$ of each task T_i to be given task properties that are constants (from an ILP point of view). Similarly, the number of processors m and the number of tasks in the task set n are considered constant. All other terms used in the ILP constraints are variables unless specified otherwise. At the core of the ILP formulation are four helper variables, which we define first.

- $A_{i,k}$: A binary decision variable that is set to 1 if and only if T_i is assigned to processor k. Since each task must be

assigned to exactly one processor, we have

$$\forall T_i : \sum_{k=1}^{m} A_{i,k} = 1. \tag{C1}$$

- $\pi_{i,p}$: A binary decision variable that is set to 1 if and only if T_i is assigned the priority p. Since each task must be assigned exactly one priority, we have

$$\forall T_i : \sum_{p=1}^{n} \pi_{i,p} = 1. \tag{C2}$$

- $V_{x,i}$: A binary decision variable that is forced to 1 if T_x and T_i are assigned to the same processor. If T_x and T_i are assigned to the same processor k, then $A_{i,k} = A_{x,k} = 1$ holds for some k. The following constraint exploits this property by forcing $V_{x,i}$ to 1 in this case:

$$\forall T_x : \forall T_i, T_x \neq T_i : \ \forall k, 1 \leq k \leq m : \\ V_{x,i} \geq 1 - (2 - A_{i,k} - A_{x,k}). \tag{C3}$$

- $X_{i,x}$: A binary decision variable that is set to 1 if and only if T_i has a higher priority than T_x. We first specify constraints to force $X_{i,x}$ to 0 if T_x has a higher priority than T_i:

$$\forall T_x : \forall T_i : \ \forall 1 \leq p \leq n-1 : \\ X_{i,x} \leq \sum_{j=p+1}^{n} \pi_{x,j} + (1 - \pi_{i,p}). \tag{C4}$$

  Constraint C4 is based on the observation that if there exist p_1 and p_2 such that $\pi_{x,p_1} = 1 \wedge \pi_{i,p_2} = 1 \wedge p_1 < p_2$, then $1 - \pi_{i,p_2} = 0$ and also $\sum_{j=p+1}^{n} \pi_{x,j} = 0$, and thus Constraint C4 reduces to $X_{i,x} \leq 0$ for $p = p_2$. To ensure that $X_{i,x}$ is set to 1 if T_x has a lower priority than T_i, we specify a constraint to enforce that for each pair of tasks T_i and T_x either $X_{i,x}$ or $X_{x,i}$ is set to 1:

$$\forall T_x : \forall T_i, T_x \neq T_i : \ X_{i,x} + X_{x,i} = 1. \tag{C5}$$

The ILP formulation incorporates Gai *et al.*'s analysis of the MSRP and enforces that under any valid ILP solution all tasks are indeed schedulable. That is, for each task T_i, the sum of the release jitter j_i and the response time R_i (an ILP variable) must not exceed the task's deadline d_i, which yields:

$$\forall T_i : \ j_i + R_i \leq d_i. \tag{C6}$$

To constrain the response time R_i, we decompose it into the following terms:

- e_i: the execution cost;
- B_i: the *arrival blocking* that a job can incur if a local lower-priority job is spinning or holding a resource;
- S_i: the direct and transitive *spin delay* that a job can incur due to itself and local higher-priority jobs busy-waiting for a global resource; and
- I_i: the *interference* that a job can incur due to local higher-priority jobs executing *non-critical* sections.

The response time of a task T_i is the sum of the above terms:

$$\forall T_i : \ R_i = e_i + B_i + S_i + I_i. \tag{C7}$$

Note that, although we specify all of these terms in our ILP formulation through constraints, we often do not use tight constraints on these terms, but rather upper or lower bounds that are sufficient for our goal of finding a valid partitioning. For instance, we impose only lower bounds on the spin time S_i of a task. As a consequence, if a solution to the ILP formulation, and hence a valid partitioning of a task set, can be found, this means that the task set under the partitioning implied by the set of $A_{i,k}$ and $\pi_{i,p}$ variables is schedulable; however, no other conclusions can be derived from other ILP variables (*e.g.*, about arrival blocking B_i or spin delay S_i) as these variables are not constrained to be accurate. Rather, they are merely constrained to be "sufficiently large" to rule out unschedulable partitionings. This exploits the observation that the ILP solver has an "incentive" to minimize each B_i, S_i, and I_i to satisfy Constraints C6 and C7; it is therefore not necessary to specify upper bounds for variables contributing to B_i, S_i, or I_i. As an analogy, in object-oriented terminology, the set of $A_{i,k}$ and $\pi_{i,p}$ variables represent the "public" interface to our ILP-based partitioning approach, whereas all other variables should be considered "private" and for ILP-internal use only.

Next, we specify constraints to model the interference I_i, which reflects delays due to preemptions by higher-priority jobs (modulo any spinning of such jobs, which is included in S_i).

A. A Lower Bound on the Maximum Interference I_i

The maximum *interference* I_i of T_i is the maximum total duration that a job of T_i cannot execute due to higher-priority jobs executing on the same processor, not counting any time that higher-priority jobs spend spinning. To constraint I_i, we first define the integer variable $H_{i,x}$ to denote the maximum number of jobs of T_x that can preempt a single job of T_i. This allows us to express the interference I_i as the sum of interference a job of T_i may incur from each other task:

$$\forall T_i : \ I_i = \sum_{T_x, Tx \neq T_i} H_{i,x} \cdot e_x. \tag{C8}$$

In a schedulable partitioning, the number of interfering jobs $H_{i,x}$ has to be non-negative and cannot exceed $\lceil (d_i + j_x)/p_x \rceil$, because $R_i \leq d_i$ and at most $\lceil (R_i + j_x)/p_x \rceil$ jobs of T_x can preempt a job of T_i [3]. This leads to the following constraint:

$$\forall T_x, T_x \neq T_i : \ 0 \leq H_{i,x} \leq \left\lceil \frac{d_i + j_x}{p_x} \right\rceil \tag{C9}$$

Further, $H_{i,x}$ has to be set to at least $(R_i + j_x)/p_x$ for local higher-priority tasks. For lower-priority and remote tasks, $H_{i,x}$ should be allowed to take the value of 0 as they do not interfere with T_i. This is achieved with the following constraint:

$$\forall T_x, T_x \neq T_i : \tag{C10}$$
$$H_{i,x} \geq \frac{R_i + j_x}{p_x} - \left\lceil \frac{d_i + j_x}{p_x} \right\rceil (1 - V_{i,x}) - \left\lceil \frac{d_i + j_x}{p_x} \right\rceil X_{i,x}$$

Next, we formalize the contribution of busy-waiting for global resources to a task's response time.

B. A Lower Bound on the Maximum Spin Delay S_i

The use of non-preemptive FIFO spin locks can cause blocking that contributes to a task's response time. This *spin time* is determined by the mapping of tasks to processors and the task parameters that characterize its resource access patterns, that is, $L_{i,q}$ and $N_{i,q}$. The spin time S_i models the *total* amount of direct and transitive delay that a job of J_i incurs due to busy-waiting carried out either by itself or any higher-priority job (by which it was preempted). The total spin time S_i can be broken down by the remote processors on which the critical section is executed that causes the spinning to occur. We let $S_{i,k}$ denote the worst-case cumulative delay incurred by any job of T_i due to critical sections on processor k. Then:

$$\forall T_i : \; S_i = \sum_{k=1}^{m} S_{i,k} \tag{C11}$$

The spin times $S_{i,k}$ can be further split into the delays due to different resources. That is, we can express $S_{i,k}$ as the sum of spin times $S_{i,k,q}$ that a job of T_i is delayed (directly or transitively) due to requests originating on processor k for l_q:

$$\forall T_i : \; \forall k, 1 \leq k \leq m : \; S_{i,k} = \sum_{q=1}^{n_r} S_{i,k,q}. \tag{C12}$$

The spin time $S_{i,k,q}$ depends on the longest critical section length of any request from processor k for l_q and the number of requests $N_{i,k}$ that T_i's job issues for l_q. Additionally, $S_{i,k,q}$ must incorporate delay through *transitive spinning*, that is, the time local higher-priority jobs spend busy-waiting for l_q while T_i's job is pending, which happens at most $\sum_{T_h \in \tau} H_{i,h} \cdot N_{h,q}$ times while a job of T_i pending. This is captured as follows:

$$\begin{aligned} &\forall T_i : \forall T_x, T_x \neq T_i : \; \forall q, 1 \leq q \leq n_r : \forall k, 1 \leq k \leq m : \\ &S_{i,k,q} \geq L_{x,q} \cdot \left(N_{i,q} + \sum_{T_h \in \tau} H_{i,h} \cdot N_{h,q} \right) \\ &\qquad - M \cdot (1 - A_{x,k}) - M \cdot A_{i,k} \end{aligned} \tag{C13}$$

In Constraint C13 above, we use the constant M to denote a numerically large constant "close to infinity." Formally, the constant M is chosen such that it dominates all other terms appearing in the ILP: $M = \max_{T_x,q}\{L_{x,q}\} \cdot n \cdot \max_{T_x,q}\{N_{x,q}\}$.

Note that specifying lower bounds on S_i (rather than using constraints to determine the exact values of S_i) is sufficient for our goal of finding a valid partitioning and priority assignment because any partitioning that is deemed schedulable assuming "too much" blocking is will still be schedulable if blocking is reduced. Next, we consider *arrival blocking*, which tasks can incur if lower-priority, co-located tasks access shared resources.

C. A Lower Bound on the Maximum Arrival Blocking B_i

A job of task T_i can incur *arrival blocking* when, upon its release, a lower-priority job running on the same processor is either executing non-preemptively or holding a local resource with a priority ceiling of at least T_i's priority. Similarly, the use of non-preemptive FIFO spin locks for global resources can cause a job of T_i to incur arrival blocking when a lower-priority job issues a request to a global resource. In this case, the lower-priority job non-preemptively spins until gaining access and then executes the request without giving T_i's job a chance to execute.

We first split the total arrival blocking B_i into the blocking times $B_{i,q}$ due requests from other tasks for each resource l_q:

$$\forall q, 1 \leq q \leq n_r : \; B_i \geq B_{i,q}. \tag{C14}$$

We then further split the per-resource arrival blocking times into blocking times due to requests for l_q from each processor k:

$$\forall l_q : \; B_{i,q} = \sum_{k} B_{i,q,k}. \tag{C15}$$

To constrain these per-resource, per-processor arrival blocking times for T_i, we first define a decision variable $Z_{i,q}$ that is set to 1 if critical sections of other tasks accessing resource l_q can cause a job of T_i to incur arrival blocking. To consider arrival blocking due to a local resource l_q, we enforce that $Z_{i,q}$ is set to 1 if T_i can incur blocking due to a local lower-priority task T_x accessing l_q and T_i's priority does not exceed l_q's ceiling:

$$\begin{aligned} &\forall q : \; \forall T_x, N_{x,q} > 0 \wedge T_x \neq T_i : \; \forall T_H, N_{H,q} > 0 : \\ &\qquad Z_{i,q} \geq 1 - (2 - V_{x,i} - V_{i,H}) - (1 - X_{i,x}) - X_{i,H}. \end{aligned} \tag{C16}$$

The latter three terms in the constraint disable it (*i.e.*, let it degenerate to $Z_{i,q} \geq 0$) if the tasks T_i, T_H and T_x are not assigned to the same processor, if T_x does not have a lower priority than T_i, or if l_q's ceiling is lower than T_i's priority, respectively. If l_q is a global resource, T_i can incur arrival blocking due to a local lower-priority task T_x using l_q. Further, if l_q is a global resource, there exists a remote task T_H using l_q. The the below constraint forces $Z_{i,q}$ to 1 in this case:

$$\begin{aligned} &\forall q : \; \forall T_x, N_{x,q} > 0 \wedge T_x \neq T_i : \; \forall T_H, N_{H,q} > 0 : \\ &\qquad Z_{i,q} \geq 1 - (1 - V_{x,i}) - V_{H,i} - (1 - X_{i,x}). \end{aligned} \tag{C17}$$

The decision variable $Z_{i,q}$ enables us to specify constraints for $B_{i,q,k}$. If l_q is a local resource, $B_{i,q,k}$ has to be set to at least the longest critical section length of any local lower-priority task for l_q, if requests for l_q can cause T_i to incur arrival blocking (*i.e.*, $Z_{i,q} = 1$). This can be expressed with the following constraint:

$$\begin{aligned} &\forall T_x : \; \forall k, 1 \leq k \leq m : \\ &B_{i,q,k} \geq L_{x,q} - L_{x,q} \cdot (1 - A_{x,k}) - L_{x,q} \cdot (1 - Z_{i,q}) \\ &\qquad - L_{x,q} \cdot (1 - A_{i,k}) - L_{x,q} \cdot X_{x,i}. \end{aligned} \tag{C18}$$

In case l_q is a remote resource and requests for l_q can cause T_i to incur arrival blocking, $B_{i,q,k}$ has to be set to at least the longest critical section length of any request for l_q from processor k:

$$\begin{aligned} &\forall T_x : \; \forall k, 1 \leq k \leq m : \\ &B_{i,q,k} \geq L_{x,q} - L_{x,q} \cdot (1 - A_{x,k}) \\ &\qquad - L_{x,q} \cdot (1 - Z_{i,q}) - L_{x,q} \cdot A_{i,k}. \end{aligned} \tag{C19}$$

Note that these bounds on $B_{i,q,k}$ constitute lower bounds on the *maximum* duration of arrival blocking rather than specifying the actual blocking incurred. To find a feasible solution, the ILP solver has an "incentive" to lower each $B_{i,q,k}$ as close to zero as possible, and Constraints C18 and C19 force $B_{i,q,k}$ to be large enough to reflect the worst-case non-preemptive and local blocking as determined by the MSRP analysis (*i.e.*, Constraints C18

and C19 ensure that $B_i \geq \max\{\beta_i^{NP}, \beta_i^{loc}\}$). Thus, for our goal of determining a valid partitioning, constraining B_i from below suffices to ensure the schedulability of a partitioning.

This concludes the derivation of our ILP formulation of the partitioning problem with spin locks. The key property of our approach is that it is optimal with regard to Gai *et al.*'s analysis of the MSRP [18]: any partitioning implied by a solution to Constraints C1–C19 also passes the MSRP schedulability analysis reviewed in Sec. III-B, and conversely, it can be shown that any task set and partitioning that pass the MSRP schedulability analysis also satisfies Constraints C1–C19. While a formal proof is omitted due to space constraints, we note that this equivalence stems from Constraint C7 matching the basic response-time recurrence, and the fact that, by construction, $B_i \geq \max\{\beta_i^{NP}, \beta_i^{loc}\}$ and $I_i + S_i \geq \beta_i^{rem} + \sum_{T_h, \pi_h < \pi_i \wedge P(T_i)=P(T_h)} \left\lceil \frac{R_i + j_h}{p_h} \right\rceil \cdot (e_h + \beta_h^{rem})$. This ensures that the ILP solution is never "optimistic" (*i.e.*, unschedulable under the MSRP analysis), while also ensuring that a schedulable task set implies a valid ILP solution. Next, we outline straight-forward extensions of our ILP formulation.

D. ILP Extensions

Our ILP formulation can be extended to incorporate system constraints that commonly arise in practice, as we show next.

1) Precedence Constraints: Task precedence constraints specify a partial temporal order among jobs that can be used to express an output-input dependency among tasks (*e.g.*, in a "pipeline" processing flow, where jobs of one task produce an output consumed by a job of second task, in which case the second job cannot start executing before the first job completed).

In our ILP formulation, precedence constraints can be incorporated in a straightforward fashion. A common approach is to encode precedence constraints as release jitter [3, 26], to model that a job waiting for input cannot be scheduled. Since we allow for release jitter in our model, precedence constraints can be incorporated seamlessly into the presented ILP formulation. For instance, to express that task T_x precedes task T_y, it suffices to add the constraint $j_y \geq R_x$. In this case, the task jitter is considered to be an ILP variable and not treated as a constant.

2) Locality Constraints: In practice, it may be necessary to avoid co-locating certain tasks. For instance, it might be desirable to enforce that replicated mission-critical tasks are not located on the same processor for higher resilience in the face of hardware faults. Such locality constraints can be incorporated with additional constraints in an intuitive way. Recall that our ILP formulation already uses a binary decision variable $V_{x,y}$ that is set to 1 if two tasks T_x and T_y are co-located. Forcing two tasks to be assigned to different processors can be achieved by simply adding the constraint $V_{x,y} = 0$.

3) Partial Specifications: Generalizing the locality constraints described previously, system designers might want to enforce a certain priority assignment (*e.g.*, because the most critical task should run at highest priority) or processor assignment (*e.g.*, because some tasks rely on a functionality only available on certain processors) for a subset of tasks. Another use case for enforcing such *partial specifications* is the *extension* of an existing application where new tasks and/or processors are added, but the priority and/or processor assignment of (some) existing tasks should remain unchanged. Similar to locality constraints, these partial specifications can be incorporated in our ILP formulation by adding constraints to enforce a particular variable assignment. For instance, forcing a task T_x to be mapped on a specific processor k and assigned a priority of y can be achieved with the constraints $A_{x,k} = 1$ and $\pi_{x,y} = 1$.

4) System Minimality: Our ILP approach can also be used to minimize the number of processors required to host a given task set. To that end, we set the $m = n$, such that a partitioning will certainly be found if the task set is feasible at all. This allows us to specify constraints to determine the highest processor ID K that is in use (*i.e.*, tasks are assigned to that partition): $\forall T_i : \forall 1 \leq k \leq n : K \geq k \cdot A_{i,k}$. The optimization objective is then to minimize K, which yields a partitioning with the smallest number of processors possible.

Note that these constraints make use of the variables we already defined in our ILP formulation. More complex partial specifications or requirements can be implemented by introducing additional variables to model application-specific properties. Such application-specific extensions to our ILP formulation do not require fundamental changes to our approach, but rather can be realized by specifying additional ILP constraints. We thus believe this to be a flexible technique well-suited to the realities of embedded systems development and optimization in practice.

Algorithm 1 Greedy Slacker Partitioning Heuristic

```
1:  for all tasks T_x order of decreasing density do
2:      C ← ∅
3:      for all processors p do
4:          s ← tryAssign(T_x, p)
5:          if s ≥ 0 then
6:              C ← C ∪ {(p, s)}
7:      if |C| = ∅ then
8:          return Failure
9:      else
10:         choose (p, b) from C such that b maximal
11:         assign T_x to processor p
```

VI. A Simple Resource-Aware Heuristic

Although the ILP-based approach yields optimal results (with regard to Gai *et al.*'s underlying analysis of the MSRP [18]), the inherent complexity of ILP-solving may render this approach impractical for large task sets. As an alternative, we present *Greedy Slacker*, a novel resource-aware heuristic for priority assignment and partitioning. While not necessarily finding partitions in all cases, on average, it results in higher schedulability than the other heuristics considered in this work.

Our heuristic, given in Algs. 1 and 2, considers all tasks in order of decreasing density. For each task, it determines the processors to which it can be assigned while maintaining schedulability of all previously assigned tasks (Alg. 1, line 5). Among the possible processors to which a task can be assigned, the processor is chosen such that the minimum slack $\min\{p_i - R_i | T_i \in U\}$ of all tasks on that processor is maximal (Alg. 1, line 10). To determine whether a task T_i can be assigned to a specific processor, the function `tryAssign`, a modified version of Audsley's optimal priority assignment scheme [4], is called. The function `tryAssign` tries to assign priorities

Algorithm 2 Function tryAssign

```
1:  function TRYASSIGN(T_x, p)
2:      temporarily assign T_x to processor p
3:      U ← all tasks assigned to processor p
4:      for priority π = |U| down to 1 do
5:          C ← tasks in U schedulable with priority π
6:          for c ∈ C do
7:              if task on other proc. unschedulable with c on p then
8:                  remove c from C
9:          if C = ∅ then
10:             return −1
11:         else
12:             T_max ← T_y ∈ C with longest period
13:             assign priority π to T_max
14:             U ← U \ T_max
15:     s ← min{p_i − r_i | T_i ∈ U}
16:     return s
```

to all tasks assigned to a given processor, starting with the lowest-possible priority. For each priority level, `tryAssign` checks whether the tasks to which no priority was assigned yet would remain schedulable under the current priority level (Alg. 2, line 5). If so, it is further checked whether this priority assignment would cause tasks assigned to other partitions to become unschedulable (Alg. 2, line 8). Among all possible assignments, the current priority level is assigned to the task with the longest period (Alg. 2, line 12). The algorithm continues until priorities are assigned to all tasks on the given processor, or no candidate task can be found for a priority level. In the latter case, T_i cannot be assigned to the given processor and the function returns a value indicating failure (Alg. 2, line 10). The function returns the minimal slack of all tasks assigned to the current processor if a priority assignment could be determined that ensures that all tasks are schedulable (Alg. 2, line 16).

Note that the presented heuristic does not include terms specific to any locking protocol, nor does it rely on parameters that need to be tuned for specific task sets. In fact, our heuristic is oblivious to the choice of locking protocol and uses an intriguingly simple greedy approach. This is possible because our heuristic aims to maximize the minimal slack among all tasks, which implicitly considers the impact of blocking due to resource sharing. Next, we evaluate runtime characteristics of our ILP-based partitioning scheme and the performance of our heuristic in comparison with prior approaches.

VII. Evaluation

In this section we explore the computational tractability of our optimal ILP-based partitioning scheme. Further, we evaluate the performance of the Greedy Slacker heuristic presented in this work and present a comparison with other resource-aware and generic bin-packing heuristics.

A. Runtime Characteristics of Optimal Partitioning

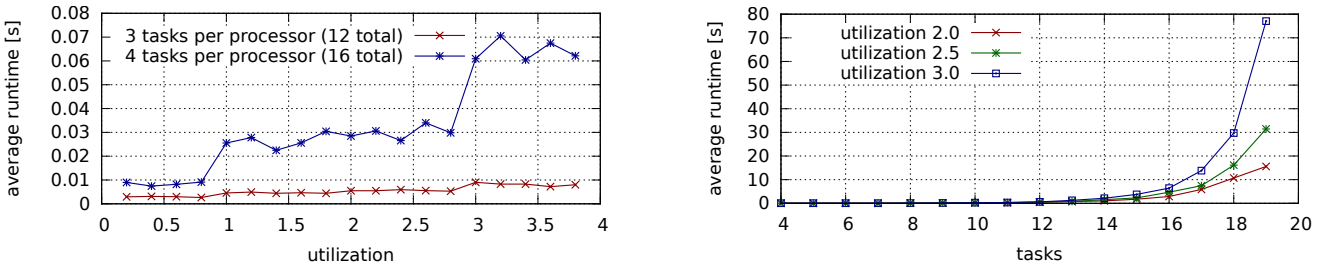
The performance of an optimal partitioning scheme in terms of schedulability is given by its definition: for each task set that *can* be partitioned such that all tasks are schedulable, an optimal partitioning scheme will find such a partitioning. Optimal partitioning approaches, however, are inherently complex which raises the question of computational tractability. We evaluated the proposed optimal ILP-based partitioning scheme in terms of average runtime depending on two key task set characteristics: total utilization and task set size. For solving the generated ILPs, we used the CPLEX 12.4 [2] optimizer running on a server-class machine equipped with 24 Intel Xeon X5650 cores with a speed of 2.66 GHz and 48 GB main memory.

In the first experiment, we measured the runtime as the total utilization of the input task sets increased. Increasing the total utilization limits the options for a valid partitioning, and hence the partitioning problem gets harder to solve. For our experiment, we assumed a multicore platform with $m = 4$ processors and evaluated task sets with 3 or 4 tasks per processor while varying the total utilization parameter. For each utilization value, 100 sample task sets were considered. The task periods are chosen at random from $[10ms, 100ms]$ according to a log-uniform distribution. Each task issues a requests for the single shared resource with a probability of 0.2. In case the shared resource is accessed, the critical section length is set to $100\mu s$. The results shown in Fig. 2a show that the runtime grows as the total utilization increases and the partitioning problem becomes harder to solve. Interestingly, the results exhibit a stepwise increase in runtime each time the total utilization approaches the next-largest integer. Further, the runtime grows rapidly as the total utilization approaches m since the partitioning problem becomes (much) harder with decreasing spare capacity.

In our second experiment, we evaluated the impact of task set size on solving time. An increase in task set size leads to a larger ILP size, and hence potentially to longer solving times. To study this effect, we fixed the total task set utilization to 2.0, 2.5 and 3.0, respectively, and varied the number of tasks in the task set from 4 to 20. Task periods and resource accesses where chosen as in the first experiment. The results are shown in Fig. 2b and exhibit a clear increase in run time as the task set is growing. Since the total utilization was kept constant, we ruled out the effect studied in the first experiment where growing utilization makes the partitioning problem harder to solve, which is reflected in higher run times. Rather, we attributed the observed effect to the growth in ILP size (in terms of the number of both constraints and variables) and resource contention, both of which increase with each additional task.

The results imply that the increase of total utilization and task set size each independently cause a significant increase in runtime of the ILP-based approach presented in Sec. V. However, the results also demonstrate that, with today's hardware, our exact ILP-based partitioning approach is applicable to small and moderate application instances (note that the runtimes reported in Figs. 2a and 2b are in the range of a couple of seconds on average). Even though run times may grow quickly for larger applications, our ILP-based partitioning technique may still be the preferred approach as it is only a one-time effort that may well be worth the cost in the context of commercial development cycles that can stretch many months or even years.

For settings where the computational complexity of the ILP-based approach is prohibitive, we proposed the resource-aware Greedy Slacker partitioning heuristics, which we evaluated with schedulability experiments, as we discuss next.



(a) Average runtimes while varying total utilization.

(b) Average runtimes while varying task set size.

Fig. 2: The average runtime costs of solving the ILP for partitioning one task in seconds.

B. Partitioning Heuristic Evaluation

For the performance comparison of our Greedy Slacker with other partitioning heuristics, we generated task sets with a broad range of configurations. We considered systems with 8 and 16 processors and 1 to 32 resources shared among the tasks. The task sets were generated using the approach presented by Emberson *et al*. [15] with periods chosen according to a log-uniform distribution from either $[3ms, 33ms]$ (*short*) or $[10ms, 100ms]$ (*moderate*) . The average per-task utilization was set to either 0.1, 0.2 or 0.3. For each configuration, we choose a *resource sharing factor* (rsf) of either 0.1, 0.25, 0.5 or 0.75, which gives the fraction of tasks accessing a given shared resource. For instance, for a task set consisting of $n = 10$ tasks, four shared resources and a sharing factor of 0.5, each of the four resources is accessed by $5 = n \cdot rsf$ tasks, which are chosen independently for each resource. For each accessed resource, only a single request is issued (*i.e.*, $N_{i,q}$) with a critical section length chosen either from $[1us, 15us]$ (*short CSLs*) or $[1us, 100us]$ (*medium CSLs*). For each data point we generated and evaluated 100 sample data sets.

We compared schedulability under the Greedy Slacker heuristic, the MPCP partitioning heuristic [20], BPA [23], and the resource-oblivious any-fit heuristic (which tries the first-, best-, next-, and worst-fit strategies, and returns the result of the first to succeed). For any-fit, we considered the following variants:

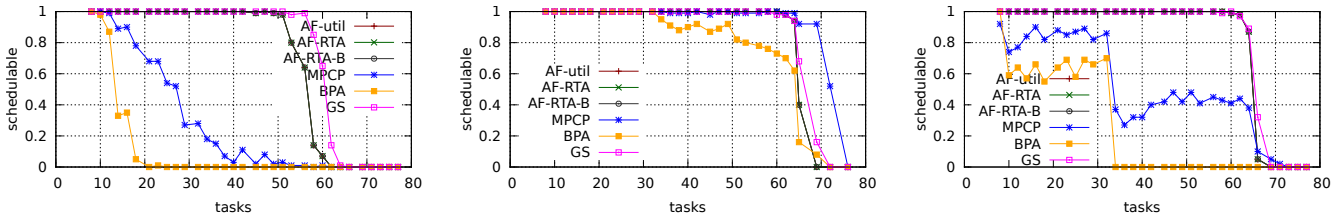
- *AF-util*: plain any-fit heuristic in which the assignment decisions of the underlying bin-packing heuristics are solely based on task utilizations (assuming a bin size of one);
- *AF-RTA*: similar to AF-util, but an additional response-time analysis is performed to rule out assignment decisions that would render a task set unschedulable immediately; and
- *AF-RTA-B*: similar to AF-RTA, but the MSRP blocking bounds are applied, so that the blocking effects due to resource sharing are considered.

Out of the large number of configurations we evaluated, we present the results for one exemplary configuration in Fig. 3a to highlight typical trends. The results of this configuration resembles trends observable in many of the configurations considered. With a growing number of tasks in each task set, both the contention for the shared resources and the total utilization increases. Up to a task set size of $n \approx 50$, AF-RTA-B is able to successfully produce valid partitionings for all task sets, but schedulability quickly drops for larger task sets. Surprisingly, AF-RTA and AF-util exhibit virtually the same schedulability as AF-RTA-B. This is due to the fact that the AF strategy applies the worst-first heuristic first, which distributes tasks roughly evenly among all cores. This benefits schedulability such that response-time and blocking checks are superfluous for most low-utilization task sets. In this particular scenario, the Greedy Slacker heuristic is able to determine valid partitionings for all task sets with up to 54 tasks, and overall Greedy Slacker achieves the highest schedulability among the considered heuristics.

Surprisingly, both the MPCP heuristic and BPA led to significantly lower schedulability than the AF. This effect was unexpected since both the MPCP heuristic and BPA were particularly designed for scenarios with resource sharing, while AF is resource-oblivious. We found that the reason for this effect lies in the way BPA and the MPCP heuristic partition task sets: both of them compute a connected component consisting of tasks that share resources (possibly transitively). For the configuration considered, this connected component is likely to include a large fraction of the task set. In this case, the MPCP heuristic and BPA attempt to break up the connected component into smaller chunks that can be fitted on a single processor such that the extent of resource sharing between these chunks is small. However, in the task sets we generated, requests to all resources are uniformly distributed over all tasks, without exhibiting a particular structure or locality among tasks and resources that could be exploited by these heuristics. The BPA and MPCP heuristics thus frequently failed to find an appropriate partitioning.

To study the performance of the MPCP heuristic and BPA when the task set exhibits some *structure* in terms of requests to shared resources, we generated task sets in which tasks are combined into *task groups*. A task group can be considered as a functional unit in a system composed of multiple tasks that share resources among them. Notably, no resources are shared across group boundaries, which results in multiple smaller connected components (one for each task group) that can be assigned to partitions without breaking them up into smaller chunks. Within each task group, tasks share the same number of resources as in the previous experiment. These resources are private to each task group, that is, different task groups share disjoint sets of resources. Fig. 3b depicts the schedulability results for task sets with the same configuration as above, but with tasks assigned to 8 disjoint task groups. The results indicate that both the MPCP heuristic and BPA can efficiently exploit this structure and yield significantly higher schedulability results than before. Further, Greedy Slacker and AF heuristics also exhibit higher schedulability in Fig. 3b than in Fig. 3a, which indicates that blocking is less of a bottleneck in this scenario.

Real applications are likely to exhibit some structure. However, tasks also often interact via resources shared across



(a) Schedulability of unstructured task sets. **(b)** Schedulability with 8 task groups. **(c)** Schedulability with 8 task groups and one cross-group resource.

Fig. 3: Schedulability for $m = 8$, 4 shared resources, medium CSLs, moderate task periods, average task utilization 0.1, and $rsf = 0.25$.

group boundaries (*e.g.*, AUTOSAR has the concept of a *virtual functional bus*, which is shared by all tasks [1]). To study the effects of cross-group resource sharing, we considered the same task-group scenario as before with the difference that a *single* resource is shared by all task groups. The results are shown in Fig. 3c. Introducing cross-group resource sharing again results in a few, large connected components that the MPCP heuristic and BPA fail to partition effectively. Notably, the Greedy Slacker heuristic yields high schedulability results independently of the structure that a task set may (or may not) exhibit, and does not depend on any protocol-specific heuristics or parameters (besides appropriate response-time analysis) The reported trends can be observed over the full range of considered configurations, which shows Greedy Slacker to be an attractive choice in a variety of scenarios, especially if it cannot be guaranteed that task sets will always exhibit a convenient structure.

VIII. Conclusion

In this work, we have considered the problem of partitioning a set of sporadic real-time tasks that share resources protected by spin locks onto a set of identical processors. Our work is motivated by the common need to minimize SWaP requirements and component costs to the extent possible. To this end, we presented an ILP-based approach for task set partitioning and priority assignment for shared-memory multiprocessor systems with shared resources. In contrast to commonly used partitioning heuristics, this approach yields optimal results (with regard to the underlying schedulability analysis) and thereby avoids over-provisioning, but is subject to high computational costs.

For cases where the cost of the ILP-based partitioning approach cannot be afforded, we presented Greedy Slacker, a novel resource-aware partitioning heuristic, which we have demonstrated to perform well on average. Greedy slacker is generic as it is neither tailored to a specific locking protocol nor dependent on task-set-specific parameter tuning, and, due to its simplicity, it is resilient in the sense that it is able to exploit locality when existent without unreasonably degrading in performance if faced with an unanticipated, ill-structured task set composition, unlike the MPCP heuristic and BPA.

In future work, it would be interesting to simplify the proposed ILP formulation to achieve higher scalability. Further, we aim to apply the Greedy Slacker heuristic to other locking protocols and scenarios such as uniform and heterogeneous multiprocessors.

References

[1] "AUTOSAR release 4.0," http://www.autosar.org, 2012.

[2] "IBM ILOG CPLEX 12.4," http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/, 2011.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.

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

[5] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, 1991.

[6] S. Baruah, "The partitioned EDF scheduling of sporadic task systems," in *Proc. RTSS*, 2011.

[7] S. Baruah and E. Bini, "Partitioned scheduling of sporadic task systems: An ILP based approach," in *Proc. DASIP*, 2008.

[8] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *Proc. RTSS*, 2005.

[9] S. K. Baruah, "Partitioning real-time tasks among heterogeneous multiprocessors," in *Proc. ICPP*, 2004.

[10] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, UNC Chapel Hill, 2011.

[11] B. Brandenburg and J. Anderson, "Optimality results for multiprocessor real-time locking," in *Proc. RTSS*, 2010.

[12] B. Chattopadhyay and S. Baruah, "A lookup-table driven approach to partitioned scheduling," in *Proc. RTAS*, 2011.

[13] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, 2011.

[14] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, 1978.

[15] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. WATERS*, 2010.

[16] N. Fisher, "The multiprocessor real-time scheduling of general task systems," Ph.D. dissertation, UNC Chapel Hill, 2007.

[17] N. Fisher and S. Baruah, "The Partitioned Scheduling of Sporadic Tasks According to Static Priorities," in *Proc. ECRTS*, 2006.

[18] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proc. RTSS*, 2001.

[19] D. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1973.

[20] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. RTSS*, 2009.

[21] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. of the ACM*, vol. 30, 1973.

[22] A. Metzner and C. Herde, "Rtsat– an optimal and efficient approach to the task allocation problem in distributed architectures," in *Proc. RTSS*, 2006.

[23] F. Nemati, T. Nolte, and M. Behnam, "Partitioning real-time systems on multiprocessors with shared resources," in *Proc. OPODIS*, 2010.

[24] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. ICDCS*, 1990.

[25] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, 1990.

[26] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and Microprogramming*, 1994.

[27] H. Zeng and M. Di Natale, "An efficient formulation of the real-time feasibility region for design optimization," *Computers, IEEE Transactions on*, vol. 62, no. 4, 2013.

[28] W. Zheng, Q. Zhu, M. D. Natale, and A. S. Vincentelli, "Definition of task allocation and priority assignment in hard real-time distributed systems," in *Proc. RTSS*, 2007.

DOCUMENT REVISION HISTORY

Date	**Description**
May 19, 2013	Conference manuscript
January 8, 2015	Replaced the term $-L_{x,q} \cdot A_{x,k}$ in Constraint C18 with $-L_{x,q} \cdot (1 - A_{x,k})$ to correct a copy&paste error.
February 12, 2015	Added the term $-M \cdot A_{i,k}$ to Constraint C13, which is required to disable the constraint if T_x and T_i are assigned to the same processor. We thank Alessandro Biondi for bringing these oversights to our attention.