Schedulability Analysis of the Linux Push and Pull Scheduler with Arbitrary Processor Affinities

Arpan Gujarati   Felipe Cerqueira   Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—Contemporary multiprocessor real-time operating systems, such as VxWorks, LynxOS, QNX, and real-time variants of Linux, allow a process to have an arbitrary processor affinity, that is, a process may be pinned to an arbitrary subset of the processors in the system. Placing such a hard constraint on process migrations can help to improve cache performance of specific multi-threaded applications, achieve isolation among components, and aid in load-balancing. However, to date, the lack of schedulability analysis for such systems prevents the use of arbitrary processor affinities in predictable hard real-time applications. In this paper, it is shown that job-level fixed-priority scheduling with arbitrary processor affinities is strictly more general than global, clustered, and partitioned job-level fixed-priority scheduling. The Linux push and pull scheduler is studied as a reference implementation and techniques for the schedulability analysis of hard real-time tasks with arbitrary processor affinity masks are presented. The proposed tests work by reducing the scheduling problem to "global-like" sub-problems to which existing global schedulability tests can be applied. Schedulability experiments show the proposed techniques to be effective.**

I. Introduction

As multicore systems have become the standard computing platform in many domains, the question of how to efficiently exploit the available hardware parallelism for real-time workloads has gained importance. In particular, the problem of scheduling multiprocessor real-time systems has received considerable attention over the past decade and various scheduling algorithms have been proposed.

One of the main dimensions along which these real-time scheduling algorithms for multiprocessors are classified is the permitted degree of migration. *Global* and *partitioned* scheduling represent two extremes of this spectrum. Under global scheduling, the scheduler dynamically dispatches ready tasks to different processors from a single queue in the order of their priorities, whereas under partitioned scheduling, each task is statically assigned to a single processor, and each processor is then scheduled independently. Researchers have also studied hybrid approaches in detail. One notable hybrid approach is *clustered scheduling*, under which processors are grouped into disjoint clusters, each task is statically assigned to a single cluster, and "global" scheduling is applied within each cluster.

Interestingly, many contemporary real-time operating systems, such as VxWorks, LynxOS, QNX, and other real-time variants of Linux, do not actually implement the schedulers as described in the literature. Instead, they use the concept of *processor affinity* to implement a more flexible migration strategy. For example, Linux provides the system call `sched_setaffinity()`, which allows the processor affinity of a process or a thread to be specified, with the interpretation that the process (or the thread) may not execute on any processor that is not part of its processor affinity. That is, processor affinities allow binding[1] a process to an arbitrary subset of processors in the system.

Processor affinities are commonly used in throughput-oriented computing to boost an application's performance (e.g., TCP throughput on multiprocessor servers [20]) and to completely isolate real-time applications from non-real-time applications by assigning them to different cores. Processor affinities can also be used to realize global, partitioned, and clustered scheduling. (E.g., to realize partitioned scheduling, each task's processor affinity is set to include exactly one processor, and to realize global scheduling, each task's processor affinity is set to include all processors.) However, what makes this feature interesting from a scheduling point of view is that *arbitrary processor affinities* (APAs) can be assigned on a task-by-task basis, which permits the specification of migration strategies that are more flexible and less regular than those studied in the literature to date.

In this paper, we present the first techniques for the schedulability analysis of sporadic tasks with APAs under *job-level fixed-priority* (JLFP) scheduling, both to enable the use of APAs in *predictable* hard real-time systems (i.e., to allow APAs to be used in systems in which the timing correctness must be established *a priori*), and to explore whether scheduling with APAs (*APA scheduling* hereafter) merits increased attention from the real-time community.

In particular, we answer three fundamental questions pertaining to APA scheduling. **(i)** Is APA scheduling strictly dominant w.r.t. global, partitioned, and clustered scheduling? (Yes, for JLFP policies, see Sec. III.) **(ii)** Is it possible to analyze APA scheduling using existing techniques for global schedulability analysis? (Yes, see Sec. IV.) **(iii)** Besides the desirable ability to isolate tasks for scheduling-unrelated reasons such as security concerns, does APA scheduling also offer improved schedulability? (To some extent, yes, see Sec. V.)

To answer these questions, we study Linux's push and pull scheduler as a reference implementation of APA scheduling and derive matching schedulability analysis.

A. Prior Work and Background

Recall that APA scheduling uses the concept of processor affinities to implement a flexible migration strategy. Therefore, we start by classifying real-time scheduling algorithms according to different migration strategies and compare them with

[1]In the context of migrations, *binding* means that a process can only migrate to (or be scheduled on) the processors that it is bound to.

APA scheduling. We then classify different priority assignment policies used in real-time scheduling and discuss how they relate to APA scheduling. Finally, we compare APA scheduling to scheduling problems of a similar structure in related domains.

According to the degree of migrations allowed, real-time scheduling algorithms either allow *unrestricted* migrations, *no* migrations, or follow a *hybrid* approach with an intermediate degree of migration. Global scheduling [14] is an example of scheduling algorithm that allows unrestricted migration of tasks across all processors (if required) while partitioned scheduling [14] is an example of a scheduling algorithm that allows no migration at all. Some notable hybrid scheduling algorithms that have been proposed include the aforementioned clustered scheduling [6, 13], *semi-partitioned* scheduling (e.g., see [1, 5, 12, 23]) and *restricted-migration* scheduling (e.g., see [1, 17]).

APA scheduling *generalizes* global, partitioned, and clustered scheduling. In other words, APA scheduling constrains each task to migrate only among a limited set of processors defined by the task's processor affinity. Therefore, using appropriate processor affinity assignment, a taskset can be modeled as a global, clustered, or partitioned taskset (see Sec. III).

Under semi-partitioned scheduling, most tasks are statically assigned to one processor (as under partitioning) and only a few tasks migrate (as under global scheduling). APA scheduling resembles semi-partitioned scheduling in that it may also allow two tasks to have separate degrees of migration. However, if and when a task migrates under APA scheduling is determined dynamically "on-demand" as under global scheduling, whereas semi-partitioned schedulers typically restrict tasks to migrate at pre-determined points in time to pre-determined processors.

APA scheduling, which restricts migrations to occur among a fixed set of processors, should also not be confused with restricted-migration scheduling. Under restricted-migration scheduling, migrations occur only at job boundaries. It limits *when* a job may migrate, whereas APA scheduling (like global, clustered, and semi-partitioned scheduling) primarily specifies *where* a job may migrate to. However, both global and semi-partitioned scheduling [1, 17] can be combined with restricted-migration scheduling and a similar approach could also be taken for APA scheduling.

Orthogonal to the degree of migration allowed, scheduling algorithms also have a choice of how to prioritize different jobs or tasks in a taskset and how these priorities may vary over time. In particular, the different priority assignment policies used in real-time scheduling can be classified either as *task-level fixed priority* (FP), job-level fixed priority (JLFP), or *job-level dynamic priority* (JLDP) policies.

A FP policy assigns a unique priority to each task, e.g., the classic *Rate Monotonic* (RM) [26] and *Deadline Monotonic* (DM) [4, 25] priority assignments fall into this category. A JLFP policy assigns a fixed priority to each job, and unlike under FP policies, two jobs of the same task may have distinct priorities; e.g., this is the case in the *Earliest Deadline First* [26] policy. A JLDP policy allows a job to have distinct priorities during its lifetime; a prominent example in this category is the *Least Laxity First* [16] policy. APA scheduling can be combined with any of these priority assignment policies.

In this paper, we restrict our focus to JLFP policies, since such policies can be implemented with low overheads [11]. However, our results pertaining to JLFP policies also apply to FP scheduling, which is the policy actually implemented in mainline Linux. Similar to priorities under a FP policy, we assume that all jobs of a task share the same processor affinity (*i.e.*, the processor affinity assignment does not vary over time).

APA scheduling could also be understood as global scheduling on a (degenerate) *unrelated heterogeneous multiprocessor* (*e.g.*, see [21]), where each task has the same, constant execution cost on any processor included in its processor affinity, and "infinite" execution cost on any other processor. However, such platforms have primarily been studied in the context of partitioned scheduling to date (e.g., see [2, 9, 21]).

Finally, the APA scheduling problem also resembles a classic non-real-time scheduling problem in which a set of non-recurrent jobs is to be scheduled on a set of *restricted identical machines* [22, 24], i.e., given a set of n jobs and a set of m parallel machines, where each job has a processing time and a set of machines to which it can be assigned, the goal is to find a schedule that optimizes a given objective (e.g., a schedule with a minimal *makespan*). However, to the best of our knowledge, this problem has not been studied in the context of the classic sporadic task model of recurrent real-time execution (or w.r.t. other recurrent task models).

B. System Model

We consider the problem of scheduling a set of n real-time tasks $\tau = \{T_1, \ldots, T_n\}$ on a set of m identical processors $\pi = \{\Pi_1, \Pi_2, \ldots, \Pi_m\}$. We adopt the classic *sporadic task model* [27], where each task $T_i = (e_i, d_i, p_i)$ is characterized by a *worst-case execution time* e_i, a *relative deadline* d_i, and a *minimum inter-arrival time* or *period* p_i. Based on the relation between its relative deadline and its period, a task T_i either has an *implicit* deadline ($d_i = p_i$), a *constrained* deadline ($d_i \leq p_i$), or an *arbitrary* deadline. The utilization u_i of a task T_i is e_i/p_i and the density δ_i of a task T_i is $e_i/\min(d_i, p_i)$.

Each task T_i also has an associated processor affinity α_i, where $\alpha_i \subseteq \pi$ is the set of processors on which T_i can be scheduled. In this initial work on the analysis of APA scheduling, we assume that α_i is static, i.e., processor affinities do not change over time. We define the joint processor affinity $cpus(\gamma)$ of a taskset γ as the set of processors on which at least one task in γ can be scheduled. Similarly, for a set of processors ρ, $tasks(\rho)$ defines the set of tasks that can be scheduled on at least one processor in ρ.

$$cpus(\gamma) = \bigcup_{\forall T_i \in \gamma} \alpha_i, \qquad tasks(\rho) = \{T_i \mid \alpha_i \cap \rho \neq \phi\} \quad (1)$$

A task T_k can (directly) *interfere* with another task T_i, i.e., delay T_i's execution, only if α_k overlaps with α_i. We let I_i denote the set of all such tasks in τ whose processor affinities overlap with α_i. In general, the exact interfering taskset depends on the scheduling policy. Therefore, we define I_i^A as the interfering taskset if T_i is scheduled under scheduling algorithm A. For example, in an FP scheduler, only higher-priority tasks can interfere with T_i. If we let $prio(T_k)$ denote

T_k's priority, where $prio(T_k) > prio(T_i)$ implies that T_k has a higher priority than T_i (i.e., T_k can preempt T_i), then

$$I_i^{FP} = \{T_k \mid prio(T_k) > prio(T_i) \wedge \alpha_k \cap \alpha_i \neq \phi\}. \quad (2)$$

For simplicity, we assume integral time throughout the paper. Therefore, any time instance t is assumed to be a non-negative integral value representing the entire interval $[t, t+1)$. We assume that tasks do not share resources (besides processors) and do not suspend themselves, i.e., a job is delayed only if other tasks interfere with it. Further, a task T_i is *backlogged* if a job of T_i is available for execution, but T_i is not scheduled on any processor. We also use two concepts frequently: *schedulability of a task* and *schedulability of a taskset*. A task $T_i \in \tau$ is schedulable on the processor platform π, if it can be shown that no job of T_i ever misses its deadline. A taskset τ is schedulable on the processor platform π if all tasks in τ are schedulable on π.

C. Paper Organization

The rest of this paper is structured as follows. In Sec. II we give a brief overview of the Linux push and pull scheduler. We also give a formal definition of an APA scheduler, assuming the Linux scheduler as a reference implementation of APA scheduling. In Sec. III, we compare APA scheduling with global, partitioned, and clustered scheduling from a schedulability perspective. In Secs. IV and V, we present schedulability analysis techniques for APA scheduling and evaluate them using schedulability experiments. Lastly, Sec. VI discusses future work and concluding remarks.

II. Push and Pull Scheduling in Linux

The Linux kernel employs an efficient scheduling framework based on processor-local queues. This framework resembles the design of a partitioned scheduler, i.e., every processor has a runqueue containing backlogged tasks and every task in the system belongs to one, and just one, runqueue. Implementing partitioned scheduling is trivial in this design by enforcing a no-migration policy (i.e., by assigning singleton processor affinities). However, the Linux scheduler is also capable of emulating global and APA scheduling using appropriate processor affinities and migrations. We review the Linux scheduler implementation of global and APA scheduling in the remainder of this section to illustrate the similarities between these two approaches, which we later exploit to derive schedulability analysis techniques for APA scheduling.

A. Global Scheduling with Push and Pull Operations

Under global scheduling, all backlogged tasks are conceptually stored in a single priority-ordered queue that is served by all processors, and the highest-priority tasks from this queue are scheduled. A single runqueue guarantees that the system is work-conserving and that it always schedules the m highest-priority tasks (if that many are available). In preparation of our analysis of APA scheduling, we summarize global scheduling as follows.

Global Scheduling Invariant: *Let $S(t)$ be the set of all tasks that are scheduled on any of the m processors at time t. Let $prio(T_i)$ denote the priority of a task T_i at time t. If T_b is a backlogged task at time t, then under global scheduling:*

$$\forall T_s \in S(t),\ prio(T_b) \leq prio(T_s) \wedge |S(t)| = m. \quad (3)$$

However, the Linux scheduler implements runqueues in a partitioned fashion. Therefore, to satisfy the global scheduling invariant, Linux requires explicitly triggered *migrations* so that a task is scheduled as soon as at least one of the processors is not executing a higher-priority task. These migrations are achieved by so-called “push” and “pull” operations, which are source-initiated and target-initiated migrations, respectively, as described next.

Let Π_s denote the source and let Π_t denote the target processor, and let T_m be the task to be migrated. A *push* operation is performed by Π_s on T_m if T_m becomes available for execution on Π_s's runqueue (e.g., when a new job of T_m arrives, when a job of T_m resumes from suspension, or when a job of T_m is preempted by a higher priority job). The push operation iterates over runqueues of all processors and tries to identify the best runqueue (belonging to the target processor Π_t) such that the task currently assigned to Π_t has a lower priority than T_m.

In contrast to a push operation, a *pull* operation is a target-initiated migration carried out by processor Π_t when it is about to schedule a job of priority lower than the previously scheduled task (e.g., when the previous job of a higher-priority task suspended or completed). The pull operation scans each processor Π_s for a task T_m assigned to Π_s's runqueue such that T_m is backlogged and T_m's priority exceeds that of all local tasks in Π_t's runqueue. When multiple candidate tasks such as T_m are available for migration, the pull operation selects the task with the highest priority.

Preemptions are enacted as follows in Linux. Suppose a processor Π_s is currently serving a low-priority task T_l when a higher-priority task T_h becomes availabe for execution on Π_s (i.e., processor Π_s handles the interrupt that causes T_h to release a job). Then Π_s immediately schedules T_h instead of T_l and invokes a push operation on T_l to determine if T_l can be scheduled elsewhere. If no suitable migration target Π_t exists for T_l at the time of preemption, T_l will remain queued on Π_s until it is discovered later by a pull operation (or until T_h's job completes and Π_s becomes available again).

Crucially, a push operation is triggered only for tasks that are *not* currently scheduled, and a pull operation similarly never migrates a task that is already scheduled. Thus, once a task is scheduled on a processor Π_t, it can only be “dislodged” by the arrival of a higher-priority task on Π_t, either due to a push operation targeting Π_t or due to an interrupt handled by Π_t. On which processor a job is released depends on the specific interrupt source (e.g., timers, I/O devices, etc.), and how the interrupt routing is configured in the multiprocessor platform (e.g., interrupts could be routed to a specific processor or distributed among all processors). We make no assumptions on which processor handles interrupts, that is, we assume that a job may be released on potentially any processor.

B. APA Scheduling

APA scheduling is similar to global scheduling in that a task may have to be migrated to be scheduled. Under global scheduling, a task is allowed to migrate to any processor in the system, whereas under APA scheduling, a task is allowed to migrate only to processors included in its processor affinity set. Therefore, APA scheduling provides a slightly different guarantee than the global scheduling invariant.

APA Scheduling Invariant: *Let T_b be a backlogged task at time t with processor affinity α_b. Let $S'(t)$ be the set of tasks that are scheduled on any processors in α_b at time t. If $prio(T_i)$ denotes the priority of a task T_i at time t, then under APA scheduling:*

$$\forall T_s \in S'(t),\ prio(T_b) \leq prio(T_s) \wedge |S'(t)| = |\alpha_b|. \qquad (4)$$

A key feature of Linux's scheduler is that push and pull operations seamlessly generalize to APA scheduling. A push operation on Π_s migrates T_m from Π_s to Π_t only if $\Pi_t \in \alpha_m$. Similarly, a pull operation on Π_t pulls T_m from Π_s only if $\Pi_t \in \alpha_m$. In short, the two operations never violate a task's processor affinity when it is migrated.

The push and pull operations together ensure that a task T_m is waiting to be scheduled only if all processors in α_m are busy executing higher-priority tasks. However, as discussed above, note that push and pull operations never migrate already scheduled, higher-priority tasks to "make room" for T_m. As a result, T_m may remain backlogged if all processors in α_m are occupied by higher-priority tasks, even if some task $T_h \in S'(t)$ could be scheduled on another processor Π_x not part of α_m (i.e., in the worst case, if $\Pi_x \in \alpha_h$ and $\Pi_x \notin \alpha_m$, then Π_x may idle while T_m is backlogged). For instance, such a scenario may occur if T_h is released on the processor that T_m is scheduled on since Linux switches immediately to higher-priority tasks and only then attempts to push the preempted task. While this approach may not be ideal from a schedulability point of view, it has the advantage of simplifying the implementation; we revisit this issue in Sec. VI.

From the definitions of the global and APA scheduling invariants, we can infer that global scheduling is a special case of APA scheduling, where all tasks have an affinity $\alpha_i = \pi$. Conversely, APA scheduling is more general than global scheduling, but also "global-like" from the point of view of backlogged task—a task is only backlogged if "all available" processors are serving higher-priority tasks. We discuss this idea in detail in the next sections. We begin by showing APA JLFP scheduling to strictly dominate global, clustered, and partitioned JLFP scheduling in Sec. III below, and then present in Sec. IV general schedulability tests applicable to all schedulers that guarantee the APA scheduling invariant given in Equation (4).

III. Generality of APA Scheduling

Recall from Sec. I that a careful assignment of processor affinities can improve throughput, can simplify load balancing (e.g., to satisfy thermal constraints), and can be used to isolate applications from each other (*e.g.*, for security reasons). In this section, we weigh the schedulability benefits of APA scheduling against global and partitioned scheduling and show that APAs are a useful construct from the point of view of real-time scheduling as well.

As discussed in Sec. I, APA scheduling is a constrained-migration model that limits the scheduling and migration of a task to an arbitrary set of processors. By assigning an appropriate processor affinity, a task can either be allowed to migrate among all processors (like global scheduling), allowed to migrate among a subset of processors (like clustered scheduling), or not allowed to migrate at all (like partitioned scheduling).

***Lemma** 1: A taskset that is schedulable under global, partitioned, or clustered scheduling, is also schedulable under APA scheduling.*

Proof-sketch: Trivial; APA scheduling can emulate global, clustered, and partitioned scheduling by assigning every task in the taskset an appropriate processor affinity. ■

However, unlike under clustered scheduling, the processor affinities of tasks under APA scheduling need not be disjoint, i.e., two tasks T_i and T_k can have non-equal processor affinities α_i and α_k such that $\alpha_i \cap \alpha_k \neq \emptyset$. As a result, there exist tasksets that are schedulable under APA scheduling, but infeasible under global, clustered, and partitioned scheduling.

***Theorem** 1: APA JLFP scheduling strictly dominates global, partitioned, and clustered JLFP scheduling.*

Proof: We show that there exists a taskset which is schedulable under APA scheduling, but not schedulable under global, partitioned, or clustered JLFP scheduling. Consider the taskset described in Table I, which is to be scheduled on two processors. Consider any JLFP rule to prioritize tasks and an asynchronous arrival sequence, where task T_2 arrives at time 1 but all other tasks arrive at time 0. We try to schedule this taskset using global, partitioned, and APA JLFP scheduling. We do not explicitly consider clustered scheduling because, for two processors, clustered scheduling reduces to either global or partitioned scheduling.

Global scheduling: Since tasks T_1 and T_2 have unit densities each and there are two processors in the system, to obtain a schedule without any deadline misses, jobs of these tasks must always have the two highest priorities (although their relative priority ordering may differ under different JLFP policies). Also, since the deadlines of tasks T_3 and T_4 are very small compared to the execution costs of tasks T_5, T_6, and T_7, jobs of tasks T_3 and T_4 must be assigned higher priorities relative to the jobs of tasks T_5, T_6, and T_7. Due to these constraints, either jobs of T_3 must be assigned the third-highest priority and jobs of T_4 the fourth-highest priority, or vice versa. In either case, either T_3 or T_4 (whichever has the job with the lower priority) misses its deadline because neither can exploit the parallelism during $[3, 4)$, as illustrated in Figure 1. Therefore, *the taskset is infeasible under global JLFP scheduling with any JLFP rule.*

Partitioned scheduling: A feasible partition must have a total utilization of at most one. The utilizations of tasks T_5, T_6, and T_7 are 0.501, 0.5001, and 0.5 respectively. Clearly, these three tasks cannot be partitioned into two bins, each with total utilization at most one. Therefore, *the taskset cannot be*

TABLE I

Task	e_i	d_i	p_i
T_1	1	1	$10,000$
T_2	2	2	$10,000$
T_3	3	4	$10,000$
T_4	2	4	$10,000$
T_5	501	$1,000$	$1,000$
T_6	$5,001$	$10,000$	$10,000$
T_7	$5,000$	$10,000$	$10,000$

TABLE II

Task	α_i
T_1	$\{\Pi_1\}$
T_2	$\{\Pi_2\}$
T_3	$\{\Pi_1\}$
T_4	$\{\Pi_2\}$
T_5	$\{\Pi_1\}$
T_6	$\{\Pi_2\}$
T_7	$\{\Pi_1, \Pi_2\}$

partitioned onto a two-processor system.

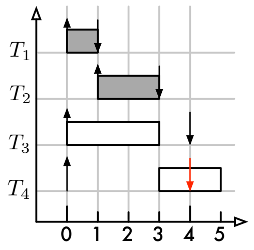
APA scheduling: The failure of global scheduling suggests that tasks T_3 and T_4 (and also tasks T_1 and T_2 because of their unit densities) should be restricted to separate processors. This separation cannot be achieved by partitioning as tasks T_5, T_6, and T_7 prevent successful partitioning of the taskset. Therefore, using processor affinities as given in Table II, we partition tasks T_1, T_2, T_3, T_4, T_5, and T_6 but allow task T_7 to migrate. The taskset is now schedulable assuming FP as the JLFP rule (lower indices imply higher priorites). To show this, we next prove the schedulability of task T_7 (tasks T_1, T_2, T_3, T_4, T_5, and T_6 can be trivially shown to be schedulable using uniprocessor response-time analysis). Consider an interval $\Gamma = [t_a, t_d)$ of length 10,000 where t_a is the arrival time and t_d is the absolute deadline of a job J_7 belonging to task T_7. We look at the two processors Π_2 and Π_1 in sequence. First, we bound the minimum time for which J_7 can execute on Π_2. Then, to ensure T_7's schedulability, we argue that J_7 can always satisfy its remaining processor demand on Π_1.

We use techniques from [7] to bound the maximum interference such that any demand due to a carry-in job (i.e., a job released prior to t_a) is also accounted for. During Γ, the maximum interference incurred by J_7 on Π_2 due to tasks T_2, T_4, and T_6 is bounded by $2 + 2 + 5001 = 5005$ (the exact interference $\mathcal{I}$ varies with the inter-arrival times of jobs). If $\mathcal{I} \leq 5000$, then J_7 can be scheduled successfully on Π_2 itself. However, if $\mathcal{I} > 5000$, J_7 must satisfy its remaining demand on Π_1, i.e., if $\mathcal{I} = 5000 + \delta$ where $\delta \in [1, 5]$, then J_7 must execute on processor Π_1 for at least δ time units.

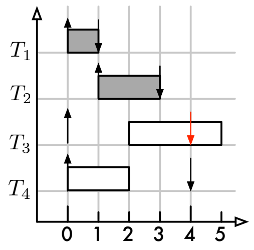
Let Γ' denote the cumulative interval(s) in Γ when jobs of T_6 interfere with J_7 on Π_2. Since the contribution of tasks T_2 and T_4 to $\mathcal{I}$ is at most $2 + 2 = 4$, the contribution of T_6 to $\mathcal{I}$ is at least $4996 + \delta$ (recall that $\mathcal{I} = 5000 + \delta$). This contribution is a result of either one job or two consecutive jobs of T_6 (in case the release of the first job of T_6 does not align with t_a but precedes t_a). In either case, Γ' consists of at least one contiguous interval $\Gamma'' \in \Gamma'$ of length $2498 + \delta/2$. However, in any contiguous interval of length $2498 + \delta/2$, Π_1 can be busy executing jobs of tasks T_1, T_3, and T_5 for at most $\lceil (2498 + \delta/2)/1000 \rceil * 501 + 2 + 2 = 1507$ time units, i.e., while Π_2 is continuously unavailable during Γ'', Π_1 is available for at least $2498 + \delta - 1507 = 991 + \delta \gg \delta$ time units, and consequently J_7 has enough opportunities to finish its remaining execution on Π_1. Therefore, T_7 is schedulable and *the taskset is schedulable under APA JLFP scheduling.*

■

Theorem 1 establishes the dominance of APA scheduling



(a) T_4's deadline miss



(b) T_3's deadline miss

Fig. 1. Global JLFP schedules of tasks T_1, T_2, T_3, and T_4 in Table I. The small up-arrows and down-arrows indicate job-arrivals and deadlines respectively. Jobs executing on processor Π_1 are in shaded in grey and jobs executing on processor Π_2 are shaded in white. **(a)** T_3's job is assigned a higher priority than T_4's job and consequently T_4's job misses its deadline. **(b)** T_4's job is assigned a higher priority than T_3's job and consequently T_3's job misses its deadline.

with a JLFP policy over global, partitioned, and clustered JLFP scheduling and provides further motivation to explore this scheduling technique. Concerning the more general case of JLDP policies, we leave the dominance question for future work. However, we note that there may be a case of equivalence since there exist global JLDP schedulers that are optimal for implicit-deadline tasks [8], and since optimal online scheduling of arbitrary-deadline tasks is generally impossible [19]. In any case, Lemma 1 shows that APA scheduling is at least as powerful as global, clustered, and partitioned scheduling combined even w.r.t. JLDP policies.

In the next section, we provide initial schedulability analysis techniques for APA schedulers with any JLFP policy.

IV. Schedulability Analysis

There are many variants of APA schedulers deployed in current real-time operating systems such as VxWorks, LynxOS, QNX, and real-time variants of Linux. However, to the best of our knowledge, no schedulability analysis test applicable to tasksets with APAs has been proposed to date. In this section, we apply the ideas from Sec. II that relate APA scheduling to the well-studied global scheduling problem, and propose simple and efficient techniques to analyze tasksets for APA scheduling. In a nutshell, we reduce APA scheduling to "global-like" subproblems, which allows reuse of the large body of literature on global schedulability analysis. The section is divided into three parts. We start with a simple method for analyzing tasksets with APAs using tests for global scheduling and argue its correctness. The second part introduces a more robust test with reduced pessimism, but at the cost of high computational complexity. The last part introduces a heuristic-based test to balance the cost versus pessimism tradeoff by considering only "promising" subproblems.

A. Reduction to Subproblems

Recall from Sections II and III that, for a given task T_i, global scheduling is a special case of APA scheduling when $\alpha_i = \pi$. Similarly, for a subproblem with a reduced processor set α_i, and a reduced taskset $tasks(\alpha_i)$, APA scheduling reduces to global scheduling. For example, consider the scheduling problems illustrated in Figure 2. Figure 2(a) represents an APA scheduling problem, where each task has an individual processor affinity. Figure 2(b) represents a subproblem of the former problem that is also an APA scheduling problem. However, as in a global scheduling problem, task T_5's processor affinity spans all the processors in this subproblem. Also, all the tasks in this subproblem can interfere with T_5. Therefore, the subproblem is global w.r.t. T_5. In other words, if T_5 is schedulable using global scheduling on a platform consisting only of the processors in α_5, then it is also schedulable using APA scheduling on the processor platform π. This idea is formally stated in the lemma below for JLFP schedulers and thus also extends to FP scheduling. Recall that $tasks(\rho)$ denotes the set of tasks that can be scheduled on at least one processor in ρ.

***Lemma** 2: If a task $T_i \in tasks(\alpha_i)$ is schedulable when the reduced taskset $tasks(\alpha_i)$ is globally scheduled on the reduced processor platform α_i using a JLFP policy A, then T_i is also schedulable under APA scheduling of τ on the processor platform π using the same JLFP policy A.*

Proof sketch: Suppose not. Then a task $T_i \in tasks(\alpha_i)$ is schedulable under global scheduling on the processor platform α_i using a JLFP policy A, but it is not schedulable under APA scheduling on the processor platform π using the same JLFP policy A. For a job J_i of any task T_i to miss its deadline, its response time r_i must be greater than its deadline, i.e., $r_i > d_i$, where r_i is the sum of T_i's WCET and the time during which J_i was interfered with by other tasks.

Task T_i incurs interference whenever all processors on which T_i can be scheduled (i.e., α_i) are busy executing tasks other than T_i. With respect to a given interval $[t_1, t_2)$, let $\Theta_i(t_1, t_2)$ denote the sub-interval (or a union of non-contiguous sub-intervals) during which all processors in α_i are busy executing tasks other than T_i. Therefore, if $|\Theta_i(t_1, t_2)|$ represents the cumulative length of the sub-intervals denoted by $\Theta_i(t_1, t_2)$, then for a job J_i arriving at t_a to miss its deadline, it is necessary that $e_i + |\Theta_i(t_a, t_a + d_i)| > d_i$.

Since T_i is not schedulable under APA scheduling on the processor platform π, there exists an arrival sequence and a corresponding interval $[t_a, t_d)$ of length d_i such that a job J_i^{APA} of T_i arrives at time t_a and misses its deadline at time t_d under APA scheduling, i.e.,

$$\exists t_a : e_i + |\Theta_i^{APA}(t_a, t_d)| > d_i. \tag{5}$$

However, since $T_i \in \tau$ is schedulable under global scheduling on the reduced processor platform α_i, for any possible arrival sequence and a corresponding interval $[t_a, t_d)$ of length d_i, a job J_i^G of T_i arriving at t_a successfully completes its execution before t_d, i.e.,

$$\forall t_a : e_i + |\Theta_i^G(t_a, t_d)| \le d_i. \tag{6}$$

The work that comprises $\Theta_i^{APA}(t_a, t_d)$ is computed upon α_i. $\Theta_i^G(t_a, t_d)$ is computed upon all processors in the processor platform, which is equal to α_i in this case. Also, by construction, under both APA and global scheduling, the same set of tasks keeps processors in α_i busy during $\Theta_i^{APA}(t_a, t_d)$ and $\Theta_i^G(t_a, t_d)$, i.e., the set of possible arrival sequences are equivalent. Therefore, if there exists an interval $[t_a, t_d)$ such that $\Theta_i^{APA}(t_a, t_d)$ exceeds $d_i - e_i$, then there exists such an interval for $\Theta_i^G(t_a, t_d)$ as well, and Equations (5) and (6) cannot both be true simultaneously. ■

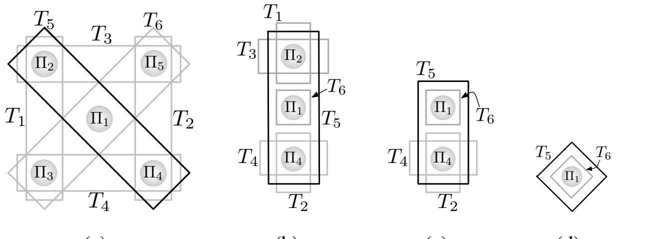


Fig. 2. Four scheduling problems (a), (b), (c), and (d) are illustrated here. The circles represent the processors and the rectangles represent the tasks and their associated processor affinities, e.g., problem (a) consists of the processor set $\pi = \{\Pi_1, \Pi_2, \Pi_3, \Pi_4, \Pi_5\}$ and the taskset $\tau = \{T_1, T_2, T_3, T_4, T_5, T_6\}$. Problem (b), (c), and (d) are subproblems of problem (a). Note that all the subproblems are global w.r.t. task T_5, i.e., like in a global scheduling problem, T_5 can be scheduled on all processors in these subproblems and all tasks in these subproblems can potentially interfere with T_5.

Using the equivalence from Lemma 2, we design a simple schedulability test for APA scheduling based on global schedulability analysis. In this paper, our focus is on global tests in general, that is, we do not focus on any particular test. For this purpose, we assume the availability of a generic test $GlobalAnalysis(A, T_i, \pi, \zeta_i)$ to analyze the schedulability of a single task, where A is the scheduling policy, T_i is the task to be analyzed, π is the processor set on which the task is to be scheduled, and ζ_i is the set of tasks that can interfere with T_i. The test returns *true* if T_i is schedulable and *false* otherwise. Note that a result of *true* does not imply that all tasks in τ are schedulable; we are only concerned with schedulability of task T_i. Using this interface we define a simple method to analyze tasks sets for APA scheduling. The key idea is to identify for each task an "equivalent" global subproblem, and to then invoke $GlobalAnalysis(A, T_i, \pi, \zeta_i)$ on that subproblem.

***Lemma** 3: A taskset τ is schedulable on a processor set π under APA scheduling using a JLFP policy A if*

$$\bigwedge_{\forall T_i \in \tau} GlobalAnalysis(A, T_i, \alpha_i, I_i^A). \tag{7}$$

Proof sketch: The analysis checks schedulability of each task $T_i \in \tau$ under global scheduling on processor platform α_i. From Lemma 2, if each task $T_i \in \tau$ is schedulable on the corresponding reduced processor platform α_i, then T_i is also schedulable on the processor platform π under APA scheduling. Therefore, the entire taskset τ is schedulable under APA scheduling with policy A on the processor platform π. ■

The analysis technique in Lemma 3 is a straightforward way to reuse global schedulability analysis for analyzing tasksets with APAs, i.e., tasksets to be scheduled by APA scheduling. Apart from the computations required by a conventional global schedulability test, this new analysis technique requires only minor additions for computing the interfering taskset (e.g. I_i^{FP} for an FP rule, I_i^{EDF} for an EDF rule) for every task T_i on the respective processor platform α_i. However, this algorithm assumes that the processors in overlapping affinity regions must service the demand of all tasks that can be scheduled in that overlapping region. Therefore, it is possible that a

TABLE III

Task	e_i	d_i	p_i	α_i
T_1	5	6	6	$\{\Pi_2, \Pi_3\}$
T_2	3	4	4	$\{\Pi_4, \Pi_5\}$
T_3	1	4	4	$\{\Pi_2, \Pi_5\}$
T_4	2	8	8	$\{\Pi_3, \Pi_4\}$
T_5	2	5	12	$\{\Pi_1, \Pi_2, \Pi_4\}$
T_6	1	3	12	$\{\Pi_1, \Pi_3, \Pi_5\}$

schedulability test claims task T_i to be not schedulable with the given processor affinity α_i, but claims it to be schedulable with a different processor affinity $\alpha'_i \subset \alpha_i$, i.e., the result of the schedulability analysis in Lemma 3 may vary for the same task if reduced to different subproblems.

***Example** 1:* Consider the taskset described in Table III, which is to be scheduled on five processors under APA scheduling. The processor affinities of the tasks are also illustrated in Figure 2(a). Assume a fixed-priority scheduler with the following priority scheme: $\forall i < k$, $prio(T_i) > prio(T_k)$. To analyze the schedulability of the taskset, we define $GlobalAnalysis(FP, T_i, \alpha_i, I_i^{FP})$ in Lemma 3 as a modified version of the response-time analysis for global fixed-priority scheduling in [10] (see Sec. V for details). Task T_5 fails the $GlobalAnalysis(FP, T_5, \alpha_5, I_5^{FP})$ test with the given processor affinity (see Figure 2(b) for the corresponding subproblem), i.e, $\alpha_5 = \{\Pi_1, \Pi_2, \Pi_4\}$. However, if applied to a different *subset* of the processor affinity (Figure 2(c)), i.e., $\alpha'_5 = \{\Pi_1, \Pi_4\}$, T_5 is deemed schedulable by the test. Note that on the processor platform α_5, all four higher priority tasks can interfere with T_5; but on the processor platform α'_5, only T_2 and T_4 can interfere with T_5. Therefore, there is a significant reduction in the total interference on T_5, and consequently the test claims T_5 to be schedulable on α'_5, but not on α_5.

In the next section, we use Example 1 to motivate an analysis technique for APA scheduling that checks schedulability of a task T_i on all possible subsets of α_i. We also argue the formal correctness of this approach by proving that schedulability of T_i on a processor platform $\alpha'_i \subset \alpha_i$ implies schedulability of T_i on the processor platform α_i.

B. Exhaustive Reduction

***Lemma** 4: If a task $T_i \in \tau$ is schedulable under APA scheduling with the processor affinity $\alpha'_i \subset \alpha_i$ and taskset τ, then T_i is also schedulable under APA scheduling with the affinity α_i and taskset τ.*

Proof sketch: We prove the lemma by contradiction, analogous to Lemma 2. Recall from the proof of Lemma 2 that $\Theta_i(t_1, t_2)$ denotes the sub-interval of interference during which all processors in α_i are busy executing tasks other than T_i. Similarly, we define $\Theta'_i(t_1, t_2)$ over all processors in α'_i. We assume that T_i is not schedulable under APA scheduling with processor affinity α_i and taskset τ but T_i is schedulable under APA scheduling with the processor affinity $\alpha'_i \subset \alpha_i$, i.e., if t_a is the arrival time of a job of T_i for an arbitrary job arrival sequence, then

$$\exists t_a : e_i + |\Theta_i(t_a, t_a + d_i)| > d_i, \tag{8}$$

$$\forall t_a : e_i + |\Theta'_i(t_a, t_a + d_i)| \leq d_i. \tag{9}$$

For any arbitrary, fixed arrival sequence, at any time instance, if all processors in α_i are busy executing tasks other than T_i, then all processors in α'_i must also be executing tasks other than T_i since $\alpha'_i \subseteq \alpha_i$. Thus, $\Theta'_i(t_a, t_a + d_i)$ is a superset ($\supseteq$) of $\Theta_i(t_a, t_a + d_i)$, and hence, $|\Theta'_i(t_a, t_a + d_i)| \geq |\Theta_i(t_a, t_a + d_i)|$: Equations (8) and (9) cannot be true simultaneously. ■

In Example 1, the schedulability test could not claim task T_5 to be schedulable with a processor affinity of α_5. However, the test claimed that the same task T_5, assuming a reduced processor affinity of $\alpha'_5 \subset \alpha_5$, is schedulable. Note that this example does not contradict Lemma 4. While the result of Lemma 4 pertains to actual schedulability under APA scheduling, the schedulability test used in Example 1 is a sufficient, but not necessary, test, which is subject to inherent pessimism, both due to the subproblem reduction and because the underlying global schedulability test is only sufficient, but not necessary, as well. Therefore, it may return negative results for tasks that are actually schedulable under APA scheduling.

We next present a schedulability analysis for APA scheduling based on Lemma 4 and the simple test in Lemma 3 that exploits the oberservation that it can be beneficial to consider only a subset of a task's processor affinity. In this method, global schedulability analysis is performed for a task $T_i \in \tau$ on all possible subsets of its processor affinity, i.e., $\forall S \subseteq \alpha_i$. The task T_i is deemed schedulable if it passes the analysis for at least one such subset S, and the taskset τ is deemed schedulable if all tasks $T_i \in \tau$ pass the test. Recall from Lemma 4 that schedulability of a task T_i under APA scheduling with processor affinity $S \subseteq \alpha_i$ implies schedulability of T_i under APA scheduling with processor affinity α_i, however, it does not require modifying the processor affinity of T_i from α_i to S. In particular, while analyzing schedulability of any task T_i, the processor affinities of other tasks remain unchanged.

***Lemma** 5: A taskset τ is schedulable on a processor set π under APA scheduling using a JLFP policy A if*

$$\bigwedge_{\forall T_i \in \tau} \Big(\bigvee_{\forall S_i \subseteq \alpha_i} GlobalAnalysis(A, T_i, S_i, I_i^A \cap tasks(S_i)) \Big). \tag{10}$$

Proof sketch: If there exists a subset $S_i \subseteq \alpha_i$ such that T_i is schedulable using global scheduling on processor platform S_i using a JLFP policy A, then from Lemma 2, T_i is also schedulable under APA scheduling with the processor affinity S_i and the policy A. From Lemma 4, since T_i is schedulable under APA scheduling with the processor affinity $S_i \subseteq \alpha_i$, T_i is also schedulable under APA scheduling with the processor affinity α_i. Therefore, if corresponding subsets exist for every task in τ, the taskset τ is schedulable on the processor set π under APA scheduling using JLFP policy A. ■

The schedulability test given by the above lemma requires iterating over potentially every subset $S \subseteq \alpha_i$. This makes the algorithm robust in the sense that it eliminates all false negatives that occur when a task T_i can be claimed to be schedulable only on a subset of its processor affinity $S \subset \alpha_i$, but not on its processor affinity α_i. However, since $|\alpha_i|$ is bounded by m, and since the schedulability tests have to be run for all the tasks in the taskset, in the worst case, the algorithm requires $O(n \cdot 2^m)$ invocations of $GlobalAnalysis(A, T_i, \alpha_i, I_i^A)$. Despite the exponential complexity, an exhaustive approach is still feasible for contemporary embedded multiprocessors with up to eight processors. However, for multiprocessor systems with higher number of processors, we need an alternative algorithm that does not analyze a task for all possible subsets of its processor affinity. Instead, in the next section, we propose a heuristic to identify and test only a few "promising" subsets.

C. Heuristic-based Reduction

We propose a heuristic that helps to choose promising subsets of a task's processor affinity to test the task's schedulability. The heuristic removes one or a few processors at a time from the task's processor affinity such that maximum benefit is achieved in terms of the interference lost (i.e., the processor time gained). We illustrate this intuition with an example below and then proceed with a detailed explanation of the heuristic and the new analysis technique.

Example 2: Consider the taskset from Example 1 (Table III). Since the schedulability of tasks $T_1, T_2, \ldots, T_5$ has already been established in Example 1, we carry out analysis for task T_6 in this example. T_6 fails $GlobalAnalysis(FP, T_6, \alpha_6, I_6^{FP})$ with the processor affinity as given in Figure 2(b), i.e, $\alpha_6 = \{\Pi_1, \Pi_3, \Pi_5\}$. Therefore, we find an appropriate subset $\alpha_6' \subset \alpha_6$ such that T_6 is claimed to be schedulable on processor platform α_6'. However, unlike the algorithm given in Lemma 5, we intelligently select only the promising subsets of α_6. In each iteration, we remove the processor that contributes the most to the total interference w.r.t. T_6.

Iteration 1: $\alpha_6 = \{T_1, T_3, T_5\}$. The removal candidates in α_6 are Π_1, Π_3 and Π_5. Removing Π_1 leads to removal of $\{T_5\}$, removing Π_3 leads to removal of $\{T_1, T_4\}$ and removing Π_5 leads to removal of $\{T_2, T_3\}$ from I_6^{FP}. We remove $\{\Pi_3\}$ because $\{T_1, T_4\}$ contributes most to the total interference on T_6. But T_6 still fails the schedulability test.

Iteration 2: $\alpha_6' = \{\Pi_1, \Pi_5\}$. The removal candidates in α_6' are Π_1 and Π_5. Removing Π_1 leads to removal of $\{T_5\}$ and removing Π_5 leads to removal of $\{T_2, T_3\}$ from I_6^{FP}. We choose to remove $\{\Pi_5\}$ because $\{T_2, T_3\}$ contributes more to the total interference on T_6 than $\{T_5\}$. The new subset is thus $\alpha_6'' = \{\Pi_5\}$ and T_6 passes the schedulability test. Therefore, T_6 is schedulable under APA scheduling with an FP policy.

The intuition of iteratively removing processors from the processor affinity until the processor set is empty is formalized with the heuristic-based algorithm $HeuristicBasedAnalysis(T_i, \alpha_i, I_i)$, which is defined in Algorithm 1. With this procedure, we obtain a new schedulability analysis for APA scheduling: a taskset τ is schedulable under APA scheduling using JLFP if $\forall T_i \in \tau$, $HeuristicBasedAnalysis(T_i, \alpha_i, I_i)$ returns $true$.

Algorithm 1 shows the pseudo-code for heuristically determining subsets of α_i and then invoking global analysis on those subsets. α_i^k is the new subset to be analyzed in the beginning of the k^{th} iteration and I_i^k is the corresponding interfering taskset. RC denotes the set of removal candidates.

Algorithm 1 $HeuristicBasedAnalysis(A, T_i, \alpha_i, I_i^A)$

```
1: $\alpha_i^0 \leftarrow \alpha_i$
2: $I_i^0 \leftarrow I_i^A$
3: $k \leftarrow 0$
4: **repeat**
5:     **if** $GlobalAnalysis(A, T_i, \alpha_i^k, I_i^k)$ is **true then**
6:         **return true**
7:     **end if**
8:     $RC \leftarrow \phi$
9:     **for all** $T_x \in I_i^k$ **do**
10:        $RC \leftarrow RC \cup \{\alpha_i^k \cap \alpha_x\}$
11:    **end for**
12:    **for all** $c \in RC$ **do**
13:        $t(c) \leftarrow tasks(\alpha_i^k) \setminus tasks(\alpha_i^k \setminus c)\}$
14:        $\Delta(c) \leftarrow \sum_{T_x \in t(c)} (\left\lceil \frac{d_i}{p_x} \right\rceil + 1) e_x$
15:    **end for**
16:    $c' \leftarrow c \in RC$ with largest $\frac{\Delta(c)}{|c|}$ (tie break using $|c|$)
17:    $\alpha_i^{k+1} \leftarrow \alpha_i^k \setminus c'$
18:    $I_i^{k+1} \leftarrow I_i^k \setminus t(c')$
19: **until** $(\alpha_i^{k+1} = \alpha_i^k) \vee (\alpha_i^{k+1} = \phi)$
```

A removal candidate is a set of processors c such that, if c is removed from α_i^k to obtain the new subset α_i^{k+1}, then there is a non-zero decrease in the total interference on T_i from tasks in I_i^{k+1} (compared to the total interference on T_i from the tasks in I_i^k). In other words, removing c from α_i^k should lead to removal of at least one task from I_i^k. Let $t(c)$ be the set of tasks removed from I_i^k if c is removed from α_i^k. To select the "best" removal candidate, we use a metric that we call *estimated demand reduction per processor*, as defined below ($\Delta(c)$ is computed in line 14 of Algorithm 1).

$$\frac{\Delta(c)}{|c|} = \frac{1}{|c|} \sum_{T_x \in t(c)} \left(\left\lceil \frac{d_i}{p_x} \right\rceil + 1 \right) e_x \qquad (11)$$

For a removal candidate c, the estimated demand reduction per processor quantifies the approximate reduction in total interference after the k^{th} iteration, if c was removed from α_i^k to obtain the new subset. The algorithm selects the removal candidate with the maximum estimated demand reduction per processor. In case of a tie between two or more candidates, we select the candidate with a smaller cardinality, e.g. among two candidates $c', c'' \in RC$ with equal demand reduction per processor, we select c' if $|c'| < |c''|$. This ensures that more processors are available for scheduling T_i with the same amount of approximate total interference. We run this procedure iteratively either until we find a successful subset or until there is no change in α_i^{k+1} w.r.t. α_i^k.

The procedure $HeuristicBasedAnalysis(T_i, \alpha_i, I_i)$ requires at most a number of iterations linear in the number of processors *m* because in every iteration at least one processor is removed from T_i's processor affinity. Therefore, after at most $|\alpha_i|$ iterations, the processor set becomes empty and the procedure terminates. The schedulability of a taskset τ requires each task $T_i \in \tau$ to be schedulable. Therefore, in the worst case, this algorithm requires $O(n \cdot m)$ invocations of $GlobalAnalysis(A, T_i, \alpha_i, I_i)$. Compared to the exhaustive technique discussed in the previous section, this algorithm is much quicker to converge to a suitable subset. However, it is a heuristic-based algorithm and may still miss out on prospective subsets that may yield positive results. In the next section, we present results of schedulability experiments to validate the efficiency of the proposed analysis.

V. Experiments and Evaluation

We ran two sets of schedulability experiments, to compare global, partitioned, and APA FP scheduling, and to compare the performance of the proposed heuristic-based schedulability analysis for APA scheduling with the exhaustive analysis. We start with the discussion of the experimental setup and then report on the observed trends.

To perform the experiments, we generated tasksets using Emberson *et al.*'s taskset generator [18]. Motivated by our study of Linux, we restricted our focus to FP scheduling algorithms. For global FP scheduling, we used the DkC priority assignment heuristic [15] and the response-time analysis for global fixed-priority scheduling given in [10], which we denote as G-FP-RTA. For partitioned FP scheduling (P-FP), we used uniprocessor response-time time analysis [3] and assigned DM priorities. Partitioning was carried out using five standard bin-packing heuristics: *worst-fit-decreasing*, *first-fit-decreasing*, *best-fit decreasing*, *next-fit-decreasing*, and *almost-worst-fit-decreasing*; a task set was claimed schedulable under P-FP if it could be succesfully partitioned using any of the heuristics and if each task in each partition passed the response-time test.

To implement $GlobalAnalysis(FP, T_i, \alpha_i, I_i^{FP})$ for APA scheduling, we used a modified version of G-FP-RTA, which we refer to as G-FP-APA. Note that the tasksets used in APA scheduling experiments were assigned processor affinities using a heuristic similar to the one discussed in Sec. IV, i.e., we started with a global assignment and allowed shrinking of the processor affinities till a schedulable partitioned assignment was found, or till a schedulable arbitrary assignment (intermediate of global and partitioned assignments) was achieved. Since optimal priority assignment for APA scheduling is still an open problem, we tried using both DkC and DM priorities with the aforementioned heuristic. Also, tasks with a singleton processor affinity set were analyzed using uniprocessor response time analysis for efficiency (instead of G-FP-RTA).

We considered two variants of G-FP-APA, the exhaustive approach based on Lemma 5 (G-FP-APAe) and the heuristic-based approach based on Algorithm 1 (G-FP-APAh). For the first set of experiments, we varied the number of processors m from 3 to 8. Herein, we focus on graphs corresponding to $m = 4$, $m = 6$, and $m = 8$. For the second set of experiments, m ranges from 3 to 5. We also varied the utilization from 0 to m in steps of 0.25 (excluding both end points). For each value of m and utilization u, we generated and tested 640 tasksets, with a number of tasks ranging from $m + 1$ to $2.5m$, in steps of 4. The periods of tasks were randomly chosen from [10ms, 100ms] following a log-uniform distribution. We summarize the main trends apparent in our results below.

Experiment 1 (G-FP-APAe vs. G-FP-RTA vs. P-FP): Each graph in Figure 3 consists of three curves, one for each of the three configurations, which represent the fraction of tasksets schedulable as a function of the total system utilization. For utilization greater than 75%, G-FP-APAe performs consistently better than P-FP, though the average improvement is modest, in the range of 0%-10%. From a schedulability point of view, we expect APA scheduling to provide the most benefit for tasksets that cannot be partitioned easily, nor are schedulable by global scheduling. However, due to the difficulty of (randomly) generating such tasksets, the experiment does not show the improvement that G-FP-APAe provides over its partitioned and global counterparts for such workloads.

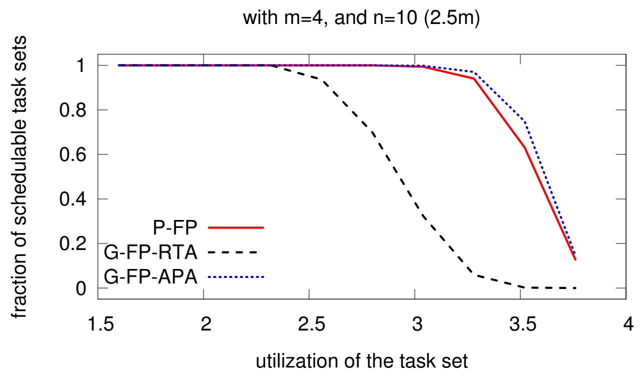
Experiment 2 (G-FP-APAh vs. G-FP-APAe): The objective of this experiment is to understand if the performance of the heuristic-based APA schedulability analysis G-FP-APAh is comparable to the exhaustive test G-FP-APAe. We used a similar experimental setup as in the first experiment, but applied both the G-FP-APAh and G-FP-APAe tests. The results in Figure 4 show that G-FP-APAh performs almost as well as to G-FP-APAe, i.e., the curves vary only slightly. This validates the efficiency of the used heuristic. Note that processor affinities were generated randomly in this experiment, which explains the overall lower schedulability compared to Experiment 1.

The experimental results demonstrate that the proposed analysis—reduction to global subproblems—is indeed effective. Further comparisons of APA scheduling with other scheduling techniques and other affinity mask assignment heuristics will certainly be interesting; however, such studies are beyond the scope of this paper and remain the subject of future work.

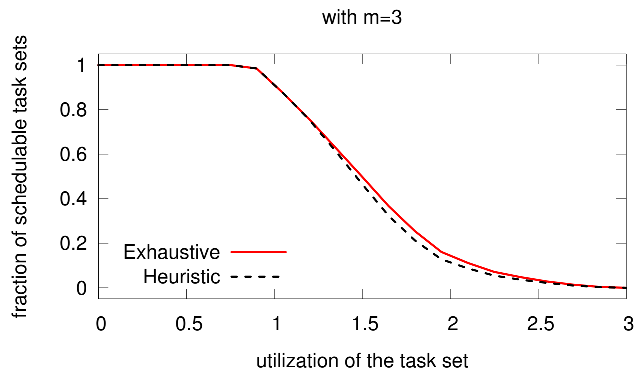
VI. Conclusion and Open Questions

In this paper, we investigated the schedulability analysis of real-time tasksets with APAs. While processor affinities have been studied and used by application developers for providing isolation and average-case enhancements, this work is the first of its kind that explores APAs from a schedulability perspective. We showed that APA-based JLFP scheduling strictly dominates global, clustered, and partitioned JLFP scheduling. The primary contribution of this paper, however, is our schedulability analysis for APA scheduling, which is simple, efficient, and reuses the extensive body of results for global scheduling already available. In summary, the paper establishes that arbitrary processor affinities are useful and can be analyzed formally. Therefore, we hope to stir further research into the design of improved analysis techniques for APA scheduling and stronger models with more flexible migration strategies.

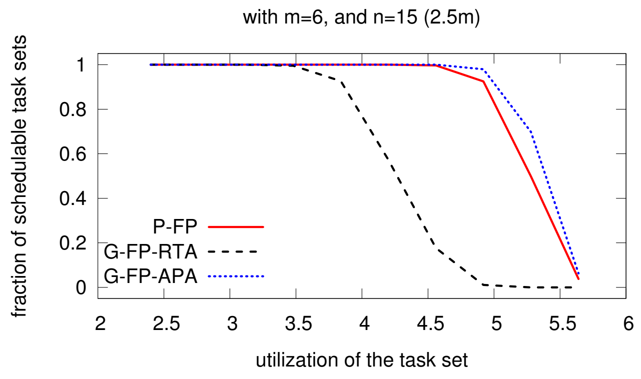
Since this paper is an initial step in the development of real-time scheduling theory for APA scheduling, there is abundant room for future work. First, a key question: is designing schedulability analysis for APA scheduling based on global schedulability analysis the best we can do? The results from our experiments are inconclusive w.r.t. this question. To answer this question, we need a feasibility test or any other schedulability test for APA scheduling that could be used as a benchmark. However, the major source of pessimism seems to be the computation of interference for affinities with high degree of overlaps, i.e., the interference computation assumes that the processors in the overlapping region have to service the entire demand of all the tasks that can be scheduled on those processors. This assumption is in some cases certainly



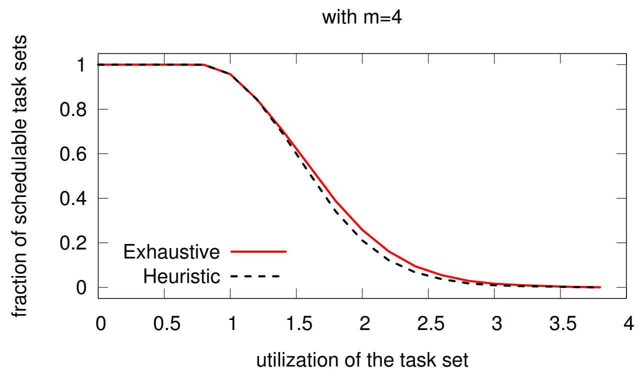
(a)



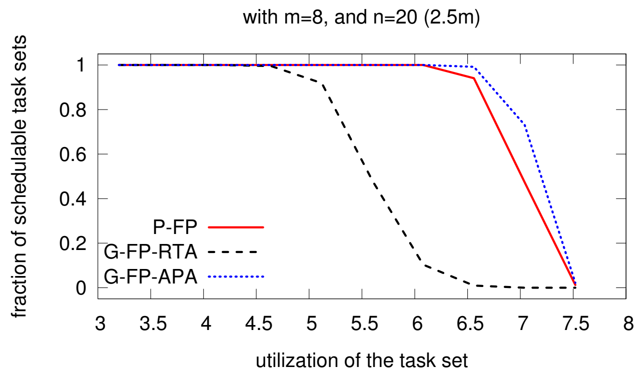
(a)



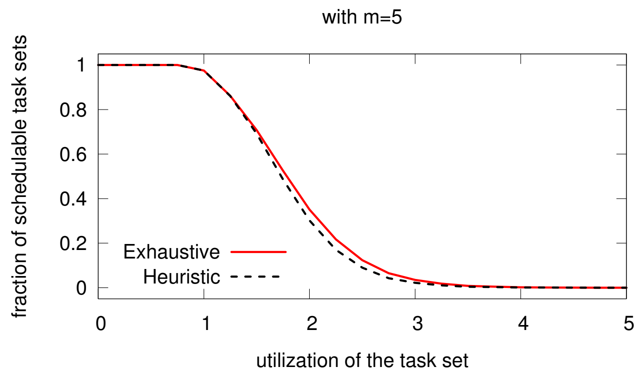
(b)



(b)



(c)



(c)

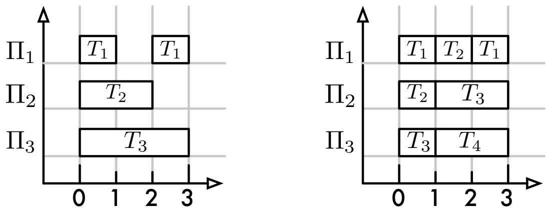
Fig. 3. The comparison of APA scheduling (G-FP-APA) versus global (G-FP-RTA) and partitioned (P-FP) scheduling.

Fig. 4. The comparison of heuristic based G-FP-RTA algorithm w.r.t. the exhaustive G-FP-RTA

inaccurate because apart from the overlapping regions, an interfering task can also be scheduled on other processors in its processor affinity. Therefore, the development of new schedulability analysis for APA scheduling from first principles would provide a useful benchmark for future improvements.

However, the assumption that tasks with overlapping affinities interfere with their entire workload is difficult to avoid in the context of *worst-case* analysis since, according to Linux's push-pull semantics, a higher-priority task interfering with a lower-priority task does not voluntarily yield the processor, even if it could migrate to an otherwise idle processor. That is, the Linux scheduler does not enact a migration if a *scheduled* higher-priority task T_h (executing on a processor Π_x) may "altruistically" give up its processor, in lieu of another processor Π_y (also in α_h), so that a lower-priority task T_l can be scheduled on Π_x (effectively *shifting* T_h from Π_x to Π_y).

For example, consider a simple taskset with implicit deadlines, which consists of tasks $T_1(1,2)$, $T_2(2,3)$, $T_3(3,4)$,



(a) Push-pull Schedule (b) Shifting Schedule

Fig. 5. The figures show the initial schedules of the taskset $T_1(1,2)$, $T_2(2,3)$, $T_3(3,4)$, and $T_4(4,12)$ having processor affinities $\{\Pi_1\}$, $\{\Pi_1,\Pi_2\}$, $\{\Pi_2,\Pi_3\}$, and $\{\Pi_3\}$ respectively, when scheduled by **(a)** Linux's push-pull scheduler and **(b)** a hypothetical shifting-migrations-based scheduler.

and $T_4(4,12)$ having processor affinities $\{\Pi_1\}$, $\{\Pi_1,\Pi_2\}$, $\{\Pi_2,\Pi_3\}$, and $\{\Pi_3\}$ respectively. Figure 5 illustrates the initial portion of two possible schedules for this taskset. Figure 5(a) shows a scheduling assuming Linux's push-pull semantics. In Figure 5(b), at time 1, T_2 shifts from Π_2 to Π_1 and T_3 shifts from Π_3 to Π_2 to enable scheduling of T_4. While in the push-pull schedule, T_4's job finishes its execution at time 12, in the second schedule, it finishes its execution at time 4. Therefore, we conjecture that a stronger migration rule like shifting will allow processor affinities to provide significantly higher schedulability, and so we seek to explore the design space of shifting-based scheduling algorithms in future work.

Finally, APAs do not place any restrictions on *when* migrations can take place. Another obvious generalization of the studied problem would be to interpret each α_i as function of time (similar to priorities), which could be used to generalize many semi-partitioned schedulers, and other hybrid schedulers, in the literature. There is also a significant room for improvements by exploring the problem of finding optimal processor affinities and optimal priority assignments.

Acknowledgement. We thank Sanjoy Baruah for pointing out the connection to the scheduling of unrelated heterogeneous multiprocessors and restricted identical machines.

References

[1] J. H. Anderson, V. Bud, and U. C. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ser. ECRTS '05. IEEE Computer Society, 2005, pp. 199–208.

[2] B. Andersson, G. Raravi, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," in *Proceedings of the 31st Real-Time Systems Symposium*, 2010, pp. 239–248.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Eng. J.*, vol. 8, no. 5, 1993.

[4] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," in *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, 1991, pp. 133–137.

[5] B. Bado, L. George, P. Courbin, and J. Goossens, "A semi-partitioned approach for parallel real-time scheduling," in *20th International Conference on Real-Time and Network Systems*. ACM, 2012, pp. 151–160.

[6] T. P. Baker and S. K. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," in *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[7] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, 2007, pp. 119 –128.

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[9] S. Baruah, "Partitioning real-time tasks among heterogeneous multiprocessors," in *Proceddings of the International Conference on Parallel Processing*, 2004, pp. 467–474.

[10] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, 2007, pp. 149–160.

[11] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[12] A. Burns, R. Davis, P. Wang, and F. Zhang, "Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme," *Real-Time Systems*, vol. 48, pp. 3–33, 2012.

[13] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 2007, pp. 247 –258.

[14] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, 2011.

[15] R. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, 2011.

[16] M. Dertouzos and A. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *Software Engineering, IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497 –1506, 1989.

[17] F. Dorin, P. M. Yomsi, J. Goossens, and P. Richard, "Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms," *CoRR*, vol. abs/1006.2637, 2010.

[18] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," *1st Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.

[19] N. Fisher, J. Goossens, and S. Baruah, "Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible," *Real-Time Systems*, vol. 45, no. 1-2, pp. 26–71, 2010.

[20] A. Foong, J. Fung, and D. Newell, "An in-depth analysis of the impact of processor affinity on network performance," in *Proceedings of the 12th IEEE International Conference on Networks*, vol. 1, 2004, pp. 244 – 250.

[21] S. H. Funk, "EDF scheduling on heterogeneous multiprocessors," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2004.

[22] J. J. Gálvez, P. M. Ruiz, and A. F. G. Skarmeta, "Heuristics for scheduling on restricted identical machines," University of Murcia, Spain, Tech. Rep., 2010.

[23] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009, pp. 249 –258.

[24] J. Y.-T. Leung and C.-L. Li, "Scheduling with processing set restrictions: A survey," *International Journal of Production Economics*, vol. 116, no. 2, pp. 251 – 262, 2008.

[25] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[26] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[27] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Massachusetts Institute of Technology, Tech. Rep., 1983.