Linux's Processor Affinity API, Refined: *Shifting* Real-Time Tasks towards Higher Schedulability

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

***Abstract*—Virtually all major real-time operating systems such as QNX, VxWorks, LynxOS, and most real-time variants of Linux expose *processor affinity* APIs to restrict task migrations. Initially motivated by throughput and isolation reasons, the ability to flexibly control migrations on a per-task basis has also proved to be useful from a schedulability perspective.**

However, as the motivation to use processor affinities is highly application-specific, the two interests can conflict, *i.e.*, the fixed, user-specified processor affinities chosen for non-schedulability reasons can actually limit any possible gains in schedulability. This paper specifically addresses the scenario where processor affinities are given as input, and investigates the following question: while maintaining API compatibility (*i.e.*, without changing the interface exposed to the programmer), is it possible to improve schedulability beyond what Linux and Linux-like systems currently offer, without violating the original affinity restrictions?

To answer this question, we explore the similarities between priority-based scheduling with processor affinities and the *assignment problem with seniority and job priority constraints*, studied previously by Caron *et al.* in an operations-research context, to derive a more generic model of migrations. Based on vertex-weighted bipartite matchings, the proposed model exploits the idea of *shifting* high-priority tasks among processors in their affinity set, in order to accommodate lower-priority tasks that have more constrained processor affinities. The proposed approach is analyzed with a novel shifting-aware schedulability analysis based on linear programming. An empirical evaluation in terms of schedulability shows shifting to be effective, although performance naturally degrades if migration overheads are high.

I. Introduction

Contemporary commodity operating systems—Linux, Windows, and OS X—as well as major real-time operating systems—QNX, VxWorks, LynxOS, *etc.*—flexibly control task migrations with processor affinity masks, which specify on a per-task basis on which processors a task may be scheduled. Although this processor-affinity API has been shown to be useful in several contexts such as application performance, fault tolerance, and security [2, 17, 18, 21, 24, 26], it is not yet well understood in the context of real-time systems.

In particular, the problem of scheduling real-time workloads with fixed priorities and *arbitrary processor affinities* (APAs) has only recently been considered [19, 20]. The initial work focused primarily on the schedulability aspects of APA schedulers, showing that, in the context of *job-level fixed-priority* (JLFP) policies, APA scheduling strictly dominates *partitioned*, *global*, and *clustered* approaches. In other words, prior work established that a careful selection of processor affinities *with the intent to improve schedulability* can indeed provide good schedulability results.

This paper, instead, is based on the assumption that processor affinities are assigned in accordance with application-specific use cases and *not* with the goal of attaining high schedulability. Given this practical scenario, where fixed restricted processor affinities are specified as input, this work explores the following question: while maintaining API compatibility (*i.e.*, without modifying the programmer-visible kernel interface), is it possible to still improve real-time schedulability without violating the user-provided processor affinity restrictions?

To answer this question, we study Linux's APA interface. Although not a hard real-time OS, we focus on Linux because it is easy to inspect and because other proprietary RTOSs such as QNX actually implement Linux-like APA scheduling semantics. We next briefly explain APA scheduling as implemented by Linux's *push* and *pull* scheduler to provide the context needed to state the contributions of this paper.

A. APAs in Linux Today: The Push and Pull Scheduler

For reasons of average-case efficiency, Linux implements per-processor run-queues, where *push* and *pull* migrations across run-queues are enacted on demand when a task arrives in the system or when a processor finishes executing a task, respectively. To conform with processor-affinity restrictions, these migrations do not move tasks outside their affinities, *i.e.*, a lower-priority ready task must wait if all processors included in its affinity mask are executing higher-priority tasks.

However, this approach of restricting the scope of migrations to only within the waiting task's processor affinity is unnecessarily restrictive, and from an analytical point of view, it does not provide the best possible schedulability. Since a lower-priority task upon its arrival can never "dislodge" a higher-priority task that could also execute elsewhere, this may needlessly prevent tasks from being scheduled, even if some processors idle as a result.

In this paper, to overcome such limitations, we explore APA semantics beyond the existing implementations and show that it is indeed possible to achieve improved schedulability (*i.e.*, lower response-time bounds) *without* violating any task's processor affinity, and *without* changing the operating system API or leaving the class of JLFP schedulers, which is highly attractive from a practical point of view.

B. This Paper: Strong APA Scheduling with Task Shifting

To this end, we relate APA scheduling to the *assignment problem with seniority constraints and job priority constraints* [12], which Caron *et al.* studied in the context of assigning employees (*i.e.*, tasks) with various levels of seniority (*i.e.*, priorities) to open positions (*i.e.*, processors), where not all employees may be suited for all positions (*i.e.*, affinity restrictions). Analogously to [12], we classify APA scheduling into *weak* and *strong* APA scheduling, which allows us to characterize Linux's push and pull scheduler more formally: it implements only weak APA scheduling.

Our primary contribution is to show that strong APA scheduling provides superior schedulability and that it can also be realized in practice by leveraging the concept of *task shifting*, *i.e.*, by allowing higher-priority tasks to be "dislodged" or moved among processors in order to make space for lower-priority tasks that are limited by affinity constraints.

To optimally decide when and which tasks to shift, we model a priority-based APA scheduler as a bipartite graph that maps tasks to processors according to affinities. By encoding priorities as vertex weights, a *maximum vertex-weighted matching* (MVM) [27] can be used to determine how tasks should be optimally assigned to processors (Sec. IV-B).

A possible downside of modeling strong APA scheduling as an MVM problem, however, is that each scheduling decision could be subject to prohibitive runtime overheads if the MVM algorithm is implemented naïvely. We therefore propose an initial online algorithm that reuses previous matchings to reduce the effort required to dispatch newly arriving tasks (Sec. IV-C).

We also propose a novel linear-programming-based *shifting*-aware schedulability analysis, which we further extend to account for the overheads of shifting migrations (Secs. V and VI). Our evaluation in Sec. VII explores the practical aspect of this work, *i.e.*, given an initial processor affinity, how large are the improvements in schedulability due to shifting? And to what extent do they outweigh the additional migration overheads?

II. Motivating Example

To motivate the potential for improved APA semantics, consider the following example, which illustrates the key ideas.

Example 1. Suppose four tasks $\{T_1, T_2, T_3, T_4\}$ are to be executed on three processors $\{\Pi_1, \Pi_2, \Pi_3\}$. Priorities are assigned in decreasing order, *i.e.*, T_1 has the highest priority. Tasks T_1, T_2, T_3, and T_4 are restricted to execute on processors $\{\Pi_1, \Pi_2, \Pi_3\}$, $\{\Pi_2, \Pi_3\}$, $\{\Pi_1\}$, and $\{\Pi_2, \Pi_3\}$, respectively. Assume that T_1, T_2, and T_4 are released at time t_1 and that T_3 is released at time $t_2 > t_1$. The initial state at time t_1 is illustrated in Fig. 1(a) using a bipartite graph where vertices correspond to tasks and processors. Initially, task T_1 is assigned to processor Π_1, T_2 is assigned to Π_2, and T_4 is assigned to Π_3. In the following, we evaluate the valid states at time t_2, that is, the state transition corresponding to task T_3's arrival.

Under the Linux scheduler, a task arrival causes a push migration, which in this case seeks to assign T_3 to either an idle processor or a processor that currently schedules a lower-priority task. However, due to T_3's affinity (it may execute only on Π_1), it must wait in Π_1's run-queue until T_1 completes. Thus, under Linux, the task-to-processor assignment at time t_2 remains as it was at time t_1 (as shown in Fig. 1(b)). Note that the lower-priority task T_4 remains scheduled even though the higher-priority task T_3 is waiting, which is an *avoidable* violation of the priority order.

If the tasks at time t_2 could be freely rearranged (*i.e.*, not confined by the limitations of a push migration), we could achieve a better schedule. In this example, tasks can be rearranged in the transition from t_1 to t_2 such that T_1, T_2, and T_3 are scheduled at time t_2 while respecting *all* affinities and priorities (as illustrated in Figs. 1(c) and 1(d)). However, Linux cannot find either of the two schedules because *shift migrations* are required in both cases. That is, to reach the state shown in inset (c), T_1 must shift from Π_1 to Π_3, and to reach the state shown in inset (d), T_1 must shift from Π_1 to Π_2, and T_2 must shift from Π_2 to Π_3. In this case, the state shown in inset (c) is preferable since it requires fewer shift migrations.

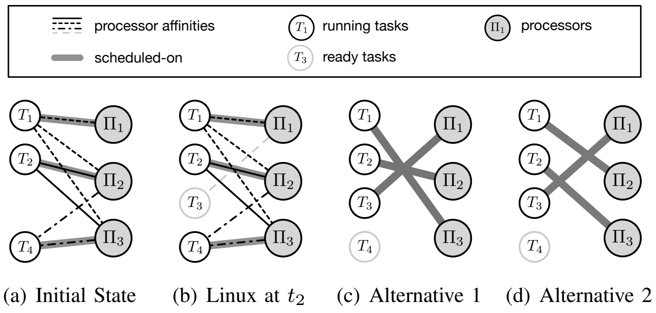


Fig. 1. Figures (a)-(d) each represents the state of the scheduler using a bipartite graph consisting of vertices corresponding to tasks and processors. **(a)** The initial state at time t_1. Task T_3 arrives at time t_2. **(b)** Linux's system state after time t_2. **(c & d)** Two alternate matchings (*i.e.*, scheduler states) that are valid at time t_2, though neither of them can be found by Linux.

To determine how to reach the desired state at time t_2, we propose a simple graph-based model and an algorithm that explores the complete state space of the system when enacting migrations, allowing for better schedulability. For example, consider the bipartite graph in Fig. 1(a) depicting the system state at time t_1. Let us associate weights with the vertices on the left so that higher-priority tasks have larger weights. Since there is no preference for a particular processor, vertices on the right side are assigned weight zero. Finding a matching of maximum vertex weight (*i.e.*, solving the MVM problem [27]) guarantees that an optimal set of tasks is assigned to the processors (Theorem 1 in Sec. IV), in the sense that no other assignment has fewer idle processors or schedules higher-priority tasks. Such a task-to-processor mapping must be recomputed every time a task arrives or departs.

In contrast to Linux's preemptions-only approach, the new approach allows shifting of higher-priority tasks in order to accommodate lower-priority tasks (as shown when T_1 shifts from Π_1 to either Π_2 or Π_3 so that T_3 can be scheduled on Π_1). Note that, at least in the absence of overheads, shifting does not affect the schedulability of the higher-priority tasks since a task that shifts continues to execute on another processor.

We formalize the ideas highlighted in this example in Sec. IV, after first establishing required definitions in Sec. III.

III. System Model and Definitions

In this section, we describe the system model and standard assumptions upon which our analysis is based, and also provide some essential definitions.

System model and assumptions: 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 $\pi = \{\Pi_1, \ldots, \Pi_m\}$. We adopt the classic *sporadic* task model [25], where each task $T_k = (e_k, d_k, p_k)$ is defined by a *worst-case execution time*

(WCET) e_k, a *relative deadline* d_k, and a *minimum inter-arrival time* or *period* p_k. Task T_k's deadline can either be *implicit* ($d_k = p_k$), *constrained* ($d_k \leq p_k$), or *arbitrary*. We assume constrained deadlines in this work. The task model is extended by associating a user-defined processor affinity α_k with every task T_k, where $\alpha_k \subseteq \pi$ is the set of processors on which T_k is allowed to be scheduled (α_k does not change over time).

In addition, the *utilization* of task T_k is defined as $u_k = e_k/p_k$ and the *utilization* of task set τ as $u_\tau = \sum_{T_k \in \tau} u_k$. The *response time* r_k of task T_k is the maximum time taken by any of T_k's jobs to complete; we denote an upper bound on r_k as r_k^{ub}. The set of *backlogged* tasks, represented as $B(t)$, consists of the tasks that are ready but not scheduled at time t.

In this work, we assume integral time, *i.e.*, any time $t \geq 0$ represents the entire real interval $[t, t+1)$. We also assume that tasks are independent and do not self-suspend. Although the schedulability analysis in Sec. V assumes negligible overheads (in particular those related to task migration), Sec. VI describes how the analysis can be extended to account for overheads.

Priority assignment: 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. While an FP policy assigns a unique priority to each task, 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. Prominent examples for FP and JLFP policies include the rate-monotonic priority assignment and the earliest-deadline-first policy, respectively [23].

Due to space constraints, the discussion of APA scheduling in Sec. IV and schedulability analysis in Sec. V are given for FP scheduling. However, the same principles can also be applied to JLFP policies without major changes. Extending the schedulability analysis to JLDP policies is left as future work. In this paper, we will denote T_k's fixed priority as $prio_k$ and the set of all tasks with priorities higher than $prio_k$ as hp_k.

Workload and interference: The *workload* $w_k(t)$ of task T_k denotes the maximum duration for which T_k can execute in a time window of length t. It is based on the number of jobs $n_k(t) = \lfloor (t + d_k - e_k)/p_k \rfloor$ that contribute with an entire WCET in this window and is formally defined as $w_k(t) = n_k(t) \cdot e_k + \min(e_k, t + d_k - e_k - n_k(t) \cdot p_k)$. The *interference* $h_i^k(t)$ of a higher-priority task T_i on the analyzed lower-priority task T_k is the cumulative length of all sub-intervals in which a job of T_k is backlogged but cannot be scheduled on any processor while T_i is executing. The interference is bounded by the workload of the interfering task and also by the latest completion time of the job of T_k, *i.e.*, $h_i^k(t) = \min(w_i(t), t - e_k + 1)$. The concepts of workload and interference are adopted from Bertogna and Cirinei [7].

Next, we formally introduce weak and strong APA scheduling and propose a graph-based strong APA scheduler.

IV. APA SCHEDULING

In Sec. II, we identified the need for a more expressive model for APA scheduling. While global and partitioned schedulers can be realized with straightforward dispatching mechanisms based on priority queues, under APA scheduling it is less obvious which tasks should execute at a given point in time.

In this section, based on a bipartite graph model (as described next in Sec. IV-A), we propose a new invariant that a correct priority-based APA scheduler should maintain. Comparing the new invariant with the APA scheduling invariant corresponding to the Linux push and pull scheduler [19], we show that the new invariant provides stronger schedulability guarantees (see Sec. IV-B). Finally, in Sec. IV-C, we provide an incremental scheduling algorithm that implements the new invariant.

A. Graph Model

Let $G(t) = (U(t) \cup V, E(t))$ denote a bipartite graph representing the scheduler state at time t. $U(t)$ is the set of ready tasks, $V = \pi$ is the set of all processors, and $E(t)$ is the set of edges representing affinities, *i.e.*, $(T_u, \Pi_v) \in E(t)$ iff $\Pi_v \in \alpha_u$. With each task $T_u \in U(t)$, we also associate a weight ϕ_u corresponding to its priority, such that tasks with higher priorities are assigned larger weights. Given this bipartite graph model, the scheduling problem at time t can be intuitively understood as finding the best possible matching $\chi(t)$ between tasks and processors, in the sense that the number of non-idle processors is maximized while maintaining the specified priority ordering, without causing any affinity violations.

B. APA Scheduler Invariants

Since tasks can be assigned in different ways, we need to properly identify an invariant representing a correct scheduler. For global scheduling, this problem is easy, as the only concern is that the m highest-priority ready tasks have to be selected. But when tasks have different affinity constraints, it is difficult to find a proper combination of assignments. In some cases, a processor may have to idle even though tasks are waiting.

To formally define a correct scheduler, we refer to Caron *et al.*'s work that addresses a conceptually equivalent problem, but in the different context of assigning employees with varying skill sets to open positions in order of seniority [12]. Most relevant to APA scheduling are Caron *et al.*'s *weak* and *strong seniority constraints*, as they are closely related to task prioritization with affinity restrictions. Below, we state analogous constraints for APA scheduling in terms of our graph model.

Invariant 1 (Weak APA Invariant).

$$\forall T_b \in B(t), \forall \pi_j \in \alpha_b, \exists T_i : (T_i, \pi_j) \in \chi(t) \wedge \phi_i \geq \phi_b. \quad (1)$$

Invariant 1 is equivalent to the weak seniority constraint defined by Caron *et al.* [12]. Intuitively, it states that for any backlogged task, all the processors in its affinity mask are occupied by higher-priority tasks. That is, there is no candidate task for preemption scheduled on any of the processors on which the backlogged task may execute.

Recall from the discussion of the Linux scheduler in Sec. II that, upon the arrival of some task T_{new}, Linux's scheduler performs only local decisions within α_{new}, searching for lower-priority tasks that can be directly preempted. Specifically, Linux never shifts higher-priority tasks in order to accommodate lower-priority tasks restricted by affinity constraints. As a result, the scheduling invariant ensured by Linux's APA scheduler (formalized in [19, 20]) is in fact equivalent to Invariant 1.

However, if shift migrations are allowed, *i.e.*, if higher-priority running tasks may be moved from one processor to another, a stronger APA scheduling invariant is possible, similar to the strong seniority constraint proposed by Caron *et al.* Before stating the strong invariant, we must define preemption and shifting migrations in terms of the graph $G(t)$.

Preemption: Given a non-assigned task T_{out}, by traversing the edges corresponding to α_{out}, every task T_j assigned to a processor $\Pi_j \in \alpha_{out}$ can be reached. By replacing a matched edge (T_j, Π_j) with (T_{out}, Π_j), we obtain a new matching $\chi'(t)$ that replaces T_j with T_{out}, corresponding to a preemption.

Shifting: The same argument can be transitively applied to paths to characterize shifting migrations. Consider an alternating path in the form $<T_{out}, \Pi_{j-k}, T_{j-k}, \ldots, \Pi_{j-1}, T_{j-1}, \Pi_j, T_j>$, where T_j is a task reachable from T_{out} via processor affinities, and every task in $\{T_{j-k}, \ldots, T_j\}$ is covered by the matching $\chi(t)$. By removing the edge (T_j, Π_j) from $\chi(t)$, adding a new edge (T_{out}, Π_{j-k}) to $\chi(t)$, and reassigning the edges $(T_{j-k}, \Pi_{j-k}) \ldots (T_{j-1}, \Pi_{j-1})$ as $(T_{j-k}, \Pi_{j-k+1}) \ldots (T_{j-1}, \Pi_j)$, respectively, we obtain a new matching $\chi'(t)$ where T_j is replaced by T_{out}.

This path-based definition can represent any sequence of migrations that respects all processor affinities, which allows us to express the strong APA invariant.

Invariant 2 (Strong APA Invariant:). *Let $R_i(t)$ denote the set of processors reachable from T_i in $G(t)$.*

$$\forall T_b \in B(t), \forall \Pi_j \in R_b(t), \exists T_i : (T_i, \Pi_j) \in \chi(t) \wedge \phi_i \geq \phi_b \qquad (2)$$

Invariant 2 corresponds to Caron *et al.*'s strong seniority constraint [12]. To illustrate the difference between the two invariants, recall Linux's scheduler state at time t_2 depicted in Fig. 1(b). With respect to the backlogged task T_3, Invariant 1 trivially holds, since the only processor on which task T_3 can execute is busy with a higher-priority task. However, note that processor Π_3 can be reached from T_3 via the neighboring task T_1. Because task T_4, which has lower priority, is executing on processor Π_3, Invariant 2 is violated. No such priority inversion occurs in the states shown in insets (c) and (d).

A mapping satisfying Invariant 2 can be obtained by solving the *maximum vertex-weighted matching* (MVM) problem, which refers to finding a bipartite matching $\chi(t)$ that maximizes the total weight $\Phi = \sum_{(T_u, \Pi_v) \in \chi(t)} \phi_u$ of the matched tasks. As shown in [27], $\chi(t)$ is also a maximum cardinality matching, which confirms that no processor unnecessarily remains idle. Maximizing Φ further guarantees that among all the matchings of maximum cardinality, $\chi(t)$ selects the highest-priority tasks. We summarize the correspondence between APA scheduling and the MVM problem in the following theorem.

Theorem 1. *An MVM in $G(t)$ satisfies Invariant 2 at time t.*

Proof. By contradiction. Let $\chi(t)$ be an MVM in $G(t)$ with total weight Φ and assume that $\chi(t)$ does not satisfy the Strong APA Invariant at time t. Then:

$$\exists T_b \in B(t), \exists \Pi_j \in R_b(t), \forall T_i : (T_i, \Pi_j) \notin \chi(t) \vee \phi_i < \phi_b. \qquad (3)$$

For such a processor Π_j in $R_b(t)$, there are two possible cases: either Π_j has an assigned task T_i such that $\phi_i < \phi_b$ or Π_j is idle. In the former case, suppose such a task T_i exists. Consider a path that leads from T_b to T_i: $<T_b, \Pi_{j-k}, T_{i-k}, \ldots, \Pi_{j-1}, T_{i-1}, \Pi_j, T_i>$. Such a path allows a shifting operation that assigns T_b and deassigns T_i, effectively increasing the total weight to $\Phi' = (\Phi - \phi_i + \phi_b) > \Phi$. Similarly, in the latter case where Π_j is idle, adding T_b to the matching leads to $\Phi' = (\Phi + \phi_b) > \Phi$. Since there is a valid edge substitution with total weight $\Phi' > \Phi$ in both the cases, $\chi(t)$ is not an MVM in $G(t)$. Contradiction. $\square$

In this paper, we refer to APA scheduling that guarantees the strong APA Invariant (*i.e.*, the MVM-based approach) as *strong APA scheduling*, and to APA scheduling that guarantees only the weak APA Invariant as *weak APA scheduling* (*e.g.*, Linux's push and pull scheduler). Having identified a correctness condition for strong APA schedulers, we now discuss one approach for how such matchings can be computed at runtime.

C. Scheduling Algorithm

Since the scheduler is a critical part of an operating system, an efficient implementation is necessary in order to avoid performance bottlenecks. For APA scheduling, this is even more important, as graph algorithms are typically more costly than the priority queues found in conventional schedulers.

A JLFP scheduler implementation has to deal with two main *scheduling events* that affect the set of ready tasks: task arrival and task departure. A correct, yet naïve, APA scheduler would recompute the MVM "from scratch" every time a scheduling event occurs. Examples of this non-incremental approach include Volgenant's algorithm [28], which improved upon Caron *et al.*'s solution [12] to achieve $O(|V| \cdot |E|)$ time complexity, and the solution described by Spencer *et al.* [27], which is based on specialized bipartite-matching algorithms and achieves $O(\sqrt{|V|} \cdot |E| \cdot \log |V|)$ time complexity (where $|V| = O(m + n)$ and $|E| = O(m \cdot n)$ denote the total number of vertices and edges in $G(t)$).

The above-mentioned algorithms perform iterations of *augmentation steps* (each with $O(|E|)$ time complexity), until convergence is reached. Although they indeed compute correct assignments, they are not ideal for APA schedulers in terms of runtime complexity. This is because an operating system is a dynamic system, where the set of ready tasks changes gradually due to scheduling events. Therefore, for performance and simplicity, it is desirable to have an incremental algorithm that, in response to a scheduling event, reuses the previous matching to compute the new task assignment.

In this work, instead of applying the multiple steps required to compute an MVM "from scratch," we propose a straightforward incremental algorithm that carries out a single $O(|E|)$ augmentation step. Although Volgenant's algorithm works in a similar way (by iteratively solving subproblems), it does not address the incremental nature of our MVM problem, in which the underlying graph changes only gradually over time.

In order to obtain a more efficient algorithm, let us consider two high-level properties of our problem that simplify the solution: **(i)** systems usually have a number of tasks that significantly exceeds the number of processors ($n \gg m$); and, as already mentioned, **(ii)** there are only two possible scheduling events, *task arrival* (*i.e.*, a job is released or resumes) and *task* *departure* (*i.e.*, a job completes or suspends). While the first property suggests that it is preferable to iterate over processors instead of tasks, the second property motivates splitting the problem into two cases (arrival and departure), which can then be solved by different, specialized algorithms. In addition, since we assume that only one task is included or removed per event, the new matching closely resembles the previous matching.

To derive an incremental algorithm, the scheduling problem can be understood in terms of state transitions. Consider the state of the scheduler at a particular time t_1, where we have a graph $G(t_1)$. Due to the occurrence of scheduling events (*e.g.*, when some task arrives), the system may move to a new state at time t_2, with a new graph $G(t_2)$. Given the previous MVM χ of total weight Φ, our objective is to find a new matching for the modified graph.

In the following, we describe how to update the matching upon task arrivals and departures. For the sake of simplicity, we assume that every idle processor is assigned an idle task T_{idle} of weight zero that is preemptable by any other task. For the pseudocode presented in Secs. IV-D and IV-E, let $\chi_\pi(T_i)$ denote the processor and $\chi_\tau(\Pi_j)$ the task for a pair $(T_i, \Pi_j) \in \chi$ corresponding to a task assignment.

D. Task Arrival

Suppose a new task T_{new} arrives at time t_2 and the scheduler had computed the previous matching χ of total weight Φ at time t_1, where $t_1 < t_2$. Since χ is an MVM in $G(t_1)$, tasks that are initially unmatched remain so in $G(t_2)$. Upon the arrival of the new task at time t_2, there are only two possible cases: either T_{new} is added to the matching for $G(t_2)$ or it is not. In the latter case, $\chi' = \chi$ maintains the weight Φ. Otherwise, we must find a better matching that includes T_{new}.

Consider every possible path in $G(t_2)$ of the form $<T_{new}, \Pi_{j-k}, T_{j-k}, \ldots, \Pi_{j-1}, T_{j-1}, \Pi_j, T_j>$. As explained in Sec. IV-B, such paths represent all feasible shifting operations that assign T_{new} and deassign some task T_j. Such an operation generates a new matching with total weight $\Phi' = \Phi + prio_{new} - prio_j$, which indicates that the scheduler must find a path that ends with the lowest-priority scheduled task T_j being preempted in favor of T_{new} (possibly an idle task).

Since we are primarily interested in the end-points (T_{new}, T_j) of the shifting paths, this corresponds to a reachability problem, which can be solved by any graph search algorithm: simply start at task T_{new} and find the reachable scheduled task T_j that has the lowest priority, which will then become unassigned. The associated path should be stored to allow backtracking at the end, in order to enact the required shifting migrations. If there is no task T_j with priority lower than T_{new}, then T_{new} is not assigned and is added to the ready queue instead.

The length of the discovered paths corresponds to the number of shifting operations that need to be performed. Thus, a *breadth-first search* (BFS) minimizes the number of migrations, as it finds a shortest path in graphs without edge weights.

The pseudocode corresponding to the arrival operation is shown in Algorithm 1. Since the graph traversal visits only matched tasks and processors, in the worst case it visits m tasks, each of which may have an affinity to all m processors

Algorithm 1 BFS algorithm for task arrival

```
1: function TASK_ARRIVAL($T_{new}$)
2:     $cpu\_to\_preempt \leftarrow nil$
3:     for each $cpu$ in $\alpha_{new}$ do
4:         Enqueue($cpu$) and mark it as visited
5:     while queue not empty do
6:         $cur \leftarrow$ Dequeue()
7:         $task\_cur \leftarrow \chi_\tau(cur)$
8:         if $task\_cur$ is the lowest priority task seen so far
9:           and $task\_cur$ has lower priority than $T_{new}$ then
10:            $cpu\_to\_preempt \leftarrow cur$
11:        if $task\_cur \neq T_{idle}$ then
12:            for each unvisited $cpu$ in $\alpha_{task\_cur}$ do
13:                Enqueue($cpu$) and mark it as visited
14:    if $cpu\_to\_preempt = nil$ then
15:        $T_{new}$ is left unassigned
16:    else Backtrack on the alternating path from $new\_task$ to
17:    $cpu\_to\_preempt$ shifting along every task, so that $new\_task$
18:    is scheduled and the task on $cpu\_to\_preempt$ is preempted.
19: end function
```

(*i.e.*, global scheduling is the worst case), for a total time complexity of $O(m^2)$ (or $O(|E|)$, assuming $m \leq n$, since $|E| = O(m \cdot n)$). The runtime for workloads with "sparse" affinities can be expected to be significantly lower.

E. Task Departure

A similar approach can be used to handle task departures. Suppose that a task T_{old} assigned to Π_{idle} completes or suspends at time t_2. Given the previous MVM χ of total weight Φ, we need to obtain a new MVM considering that T_{old}'s departure leaves processor Π_{idle} unmatched.

For departures, there are also only two possible cases. Either T_{old} is replaced by another task in the new mapping, or the processor stays idle. Consider a path in the form $<T_{old}$, Π_{idle}, T_{j-k}, Π_{j-k}, $\ldots$, T_j, $\Pi_j, T_{out}>$, obtained by following assigned tasks up to some processor Π_j that belongs to the affinity of a non-assigned task T_{out}. The corresponding task substitution produces a new matching of weight $\Phi' = \Phi - prio_{old} + prio_{out}$, where T_{out} shifts in to replace T_{old}.

To find the best replacement T_{out}, we start a BFS at Π_{idle} looking for a reachable non-assigned task of highest priority. This leads to a shifting operation that replaces T_{old} by T_{out} and maximizes the resulting weight Φ'. If the high-priority task is found in the affinity of Π_{idle}, this corresponds to a direct assignment without shifting. The pseudocode for handling a task departure is shown in Algorithm 2.

The task departure algorithm traverses every processor with matched tasks that is reachable from Π_{idle}. Following the processor affinities, it searches for the highest-priority non-assigned task, which becomes the endpoint of the chain of tasks shifting into Π_{idle}. Though Algorithm 2 may visit all $O(n \cdot m) = O(|E|)$ edges in the bipartite graph (in the case of global affinities), it improves upon the non-incremental algorithms mentioned in Sec. IV-C, which require at least $O(\sqrt{|V|} \cdot |E| \cdot \log |V|)$ steps to compute an MVM "from scratch". Again, the runtime is expected to be significantly lower for sparse graphs.

Algorithm 2 BFS algorithm for task departure

```
function TASK_DEPARTURE($idle\_cpu$)
    $task\_to\_pull \leftarrow nil$
    Enqueue($idle\_cpu$) and mark it as visited
    while queue not empty do
        $cur \leftarrow$ Dequeue()
        for each $task$ such that $cur \in \alpha_{task}$ do
            $assigned\_cpu \leftarrow \chi_\pi(task)$
            if $assigned\_cpu = nil$ then
                if $task$ has highest priority seen so far then
                    $task\_to\_pull \leftarrow task$
            else Enqueue($assigned\_cpu$) and mark it as visited
    if $task\_to\_pull = nil$ then
        $idle\_cpu$ remains idle
    else Backtrack on the alternating path from $idle\_cpu$ to
    $task\_to\_pull$, shifting along every task. Then, $idle\_cpu$
    receives a task and $task\_to\_pull$ is also scheduled.
end function
```

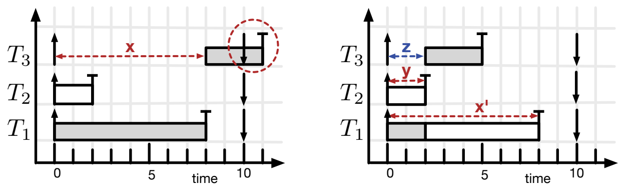
V. Schedulability Analysis

Irrespective of whether strong or weak APA semantics are used for scheduling, tasks with restricted processor affinities are (generally) difficult to analyze when their affinities overlap. To incorporate irregular interference patterns and to account for shifting, we propose a novel LP-based response-time analysis for strong APA scheduling, which we motivate with a simple example. We initially assume negligible overheads and propose techniques to account for overheads in Sec. VI. We begin with a simple example to motivate the analysis.

Example 2. Consider a task set consisting of three tasks, $\{T_1, T_2, T_3\}$, to be scheduled on two processors, $\{\Pi_1, \Pi_2\}$. Task priorities are defined in decreasing order of their indices, *i.e.*, T_1 has the highest priority. Their WCETs are 8, 2, and 3 time units, respectively; and the affinities are $\alpha_1 = \{\Pi_1, \Pi_2\}$, $\alpha_2 = \{\Pi_2\}$, and $\alpha_3 = \{\Pi_1\}$. Tasks T_1, T_2, and T_3 arrive synchronously at time $t = 0$. We discuss the schedule of these tasks in the interval $[0, 12)$ under both weak and strong APA scheduling. The resulting schedules are illustrated in Fig. 2.

The schedule under weak APA scheduling is shown in Fig. 2(a). Suppose that tasks T_1 and T_2 are respectively scheduled on processors Π_1 and Π_2 upon release. Since T_3 can only execute on Π_1, it must wait for T_1's job to complete, *i.e.*, T_1 interferes with T_3 for a total duration of $x = 8$ time units (interval $[0, 8)$). Hence, T_3's job misses its deadline at time $t = 10$. *We observe that under weak APA scheduling, the entire cost of a higher-priority job may contribute towards the interference experienced by a lower-priority job (if their affinities intersect).*

The schedule under strong APA scheduling is shown in Fig. 2(b). Unlike under weak APA scheduling, T_3 has to wait only for $z = 2$ time units during the interval $[0, 12)$. Interference on T_3 reduces in comparison to the previous scenario because at time $t = 2$, when T_2's job completes its execution, T_1 shifts from Π_1 to Π_2 and T_3 is scheduled on Π_1. Consequently, T_3's job does not miss its deadline. *In general, under strong APA scheduling, the interference of a higher-priority task on a lower-priority task (if their affinities intersect) is also indirectly upper-bounded by the duration during which other tasks prevent the higher-priority task from shifting.*



(a) Weak APA scheduling (b) Strong APA scheduling

Fig. 2. Figures (a) and (b) illustrate the execution sequence of T_1, T_2, and T_3 on Π_1 and Π_2 in the interval $[0, 12)$ under weak and strong APA scheduling. Arrows pointing upwards denote job releases, arrows pointing downwards denote deadlines, and arrows with blunt ends denote job completion. Grey (respectively, white) rectangles denote execution on processor Π_1 (respectively, Π_2). In (a), T_3's job suffers $x = 8$ units of interference and misses its deadline (dashed circle), whereas in (b), T_3's job suffers less interference ($z = 2$ units) due to T_1's shifting, and completes its execution before its deadline.

In this particular example, under strong APA scheduling, T_3 must wait only when T_1 cannot shift to any other processors, *i.e.*, when both Π_1 and Π_2 are busy. Therefore, T_1's effective interference on T_3 during the interval $[0, 12)$ is actually also bounded by the workload of T_2. In Fig. 2(b), x' denotes the workload of T_1, z denotes the interference from T_1 incurred by T_3 (*i.e.*, the total time that T_3 cannot be scheduled due to T_1's execution), and y denotes the duration that T_2 prevents T_1 from shifting in favor of T_3. We observe that trivially $z \leq x'$ (since T_1 must execute to interfere), but also that $z \leq y$, which demonstrates the key advantage of strong APA scheduling: if higher-priority tasks can shift, then their full execution time need not contribute to the interference on lower-priority tasks.

In the LP-based schedulability analysis that we introduce next, we focus on FP scheduling, but note that the same techniques can be used to analyze JLFP scheduling as well.

A. LP-based Schedulability Analysis for APA Scheduling

Many multiprocessor schedulability analyses rely on the concept of interference (as defined in Sec. III). APA schedulability analysis, however, requires a more expressive concept. Since overlapping affinities cause a task's interference to vary across processors (unlike under global scheduling), we define the concept of *per-processor interference*, denoted as $x^k_{i,p}(t)$. It denotes, in a time window of length t, the cumulative execution time of T_i on Π_p, while T_k is backlogged.

Since the exact values of per-processor interferences on T_k cannot be easily determined, we model the problem of determining $x^k_{i,p}(t)$ as a linear program (LP). The objective of the LP is to maximize r^{ub}_k, *i.e.*, the upper bound on T_k's response time r_k, given some constraints on task interference. Lemmas 1 and 2 (adapted from [20]) define constraints that we use to formulate the LP. Proofs can be found in [20]. Recall from Sec. III that $h^k_i(t)$ denotes the *interference* of T_i on T_k in a time window of length t.

Lemma 1. *In a time window of length t, if task T_k is analyzed under FP APA scheduling and task T_i has a higher priority, $\sum_{\Pi_p \in \alpha_i} x^k_{i,p}(t) \leq h^k_i(t)$ and $\forall \Pi_p \notin \alpha_i : x^k_{i,p}(t) = 0$.*

Lemma 2. *In a time window of length t, if task T_k is analyzed under FP APA scheduling and if t is the largest value such that*

$\forall \Pi_p \in \alpha_k$, $t \leq e_k + \sum_{T_i \in hp_k} x^k_{i,p}(t)$, then t is also an upper bound on the response time of task T_k.

We now state the LP that derives an upper bound on the response time of a task T_k in a time window of length t. The solution of the LP is recomputed for growing interval lengths (up to d_k) in a fixed-point iteration. If the iteration converges to a value $r^{ub}_k \leq d_k$, then r^{ub}_k is an upper bound on the response time of any job of task T_k (assuming that no higher-priority task misses a deadline). The LP variables $X^k_{i,p}$ and R^{ub}_k correspond to $x^k_{i,p}(t)$ and r^{ub}_k, respectively.

Theorem 2. *Given a task T_k to be scheduled under FP APA scheduling (either weak or strong), if no task misses a deadline, a safe response-time bound r^{ub}_k is given by the least fixed point of the equation $r^{ub}_k = R^{LP}_k(r^{ub}_k)$, starting with $r^{ub}_k = e_k$, where $R^{LP}_k(t)$ is the solution of the following LP.*

$$R^{LP}_k(t) \triangleq \textit{maximize } R^{ub}_k \textit{ subject to}$$

$$\forall T_i \in hp_k : \sum_{\Pi_p \in \alpha_i} X^k_{i,p} \leq h^k_i(t) \qquad \text{(Constraint 1)}$$

$$\forall T_i \in hp_k, \forall \Pi_p \notin \alpha_i : X^k_{i,p} = 0 \qquad \text{(Constraint 2)}$$

$$\forall \Pi_p \in \alpha_k : R^{ub}_k \leq e_k + \sum_{T_i \in hp_k} X^k_{i,p} \qquad \text{(Constraint 3)}$$

Constraints 1, 2 and 3 in Theorem 2 can be trivially derived from Lemmas 1 and 2 (see [20]). We next propose additional constraints, extending the LP in Theorem 2 to derive a shifting-aware schedulability analysis for strong APA scheduling.

B. Shifting-aware Analysis for Strong APA Scheduling

From the definition of shifting migration in Sec. IV-B, it follows that under strong APA scheduling, a reachable higher-priority task may (positively) affect the interference on a lower-priority task even if their affinities do not overlap, by virtue of shifting. Therefore, to develop a shifting-aware schedulability analysis for strong APA scheduling, we must define a bound on task interference that also reflects the workloads of non-interfering higher-priority tasks.

For instance, in the analysis of T_3's schedulability under weak APA scheduling in Ex. 2, the constraint $z \leq x'$ is trivial to find (z is the interference incurred by T_3 due to T_1 and x' denotes the workload of T_1). In contrast, strong APA scheduling also allows for an additional, less easily identifiable constraint $z \leq y$ (y denotes the duration for which T_2 prevents T_1 from shifting in favor of T_3), which restricts the total interference on T_3 to a more accurate value than under weak APA scheduling.

In order to formalize such interference relations, we define a static graph G' representing the entire task set, which is used to derive additional shifting-aware constraints for the LP.

Let $G' = (V', E')$ represent an undirected graph, where vertices in V' correspond to tasks. There exists an edge in E' between tasks T_a and T_b iff their processor affinities intersect, *i.e.*, $\alpha_a \cap \alpha_b \neq \emptyset$. In addition, let $RT_k(l)$ denote the set of tasks that are reachable from T_k in exactly l hops, *i.e.*, $T_i \in RT_k(l)$ iff there exists a path $<T_{k0}, T_{k1}, \ldots, T_{kl}>$ in G' such that $T_{k0} = T_k$ and $T_{kl} = T_i$. We further define the set $RP_k(l)$ of processors reachable in l steps as $RP_k(l) = \bigcup_{T_i \in RT_k(l)} \alpha_i$.

The following theorem considers, w.r.t. the task T_k under analysis, every shifting path of length l from T_k to a reachable processor Π_p that some task T_i can shift to, as long as no higher-priority task $T_j \in hp_k$ is executing on Π_p.

Theorem 3. *Given a time window of length t and a task T_k to be analyzed under strong FP APA scheduling, $\forall l \in \{1, \ldots, m-1\}$, $\forall T_i \in RT_k(l)$, and $\forall \Pi_p \in \alpha_i \setminus RP_k(l-1)$, the following invariant holds:*

$$\sum_{\Pi_r \in \alpha_i \cap RP_k(l-1)} x^k_{i,r}(t) \leq \sum_{T_j \in hp_k \wedge T_j \neq T_i} x^k_{j,p}(t). \qquad (4)$$

Proof. By contradiction. Assume that $\exists l \in \{1, ..., m-1\}$, $\exists T_i \in RT_k(l)$, and $\exists \Pi_p \in \alpha_i \setminus RP_k(l-1)$, such that:

$$\sum_{\Pi_r \in \alpha_i \cap RP_k(l-1)} x^k_{i,r}(t) > \sum_{T_j \in hp_k \wedge T_j \neq T_i} x^k_{j,p}(t). \qquad (5)$$

Eq. 5 implies that, in some interval of length t, the total execution time of task T_i on processors in $\alpha_i \cap RP_k(l-1)$ (while T_k is backlogged) exceeds the total execution of all other higher-priority tasks on Π_p (while T_k is backlogged). Let δ be defined as follows:

$$\delta = \sum_{\Pi_r \in \alpha_i \cap RP_k(l-1)} x^k_{i,r}(t) - \sum_{T_j \in hp_k \wedge T_j \neq T_i} x^k_{j,p}(t),$$

and let Δ represent the cumulative interval corresponding to these δ time units. Note that since $x^k_{i,r}(t)$ and $x^k_{j,p}(t)$ both denote execution time *while T_k is backlogged*, it is the case that T_k is backlogged at all times in the cumulative interval Δ.

Because δ is non-zero, there exists a point in time in Δ such that T_i is scheduled on a processor $\Pi_r \in \alpha_i \cap RP_k(l-1)$ while T_k is backlogged and none of the tasks in hp_k are scheduled on Π_p. However, in this case, T_i can shift to Π_p, thus vacating a processor in $\alpha_i \cap RP_k(l-1)$, and another higher-priority task in $RT_k(l-1)$ can shift to the processor vacated by T_i. By recursively applying this argument up to $l = 1$, we observe that T_k is scheduled after l shifts.

Assuming that shifting migrations are enacted instantaneously (*i.e.*, negligible overheads for task migrations), this implies that task T_k is not backlogged at all times during the cumulative interval Δ. Contradiction. ☐

The invariants in Theorem 3 can be used as an additional set of constraints in the LP in Theorem 2 to obtain shifting-aware response-time bounds under strong APA scheduling. Despite the larger number of constraints in the LP, in our experiments considering up to 16 processors, this analysis approach still exhibited good runtime performance (*i.e.*, the analysis required only a few seconds to complete for reasonable task set sizes). An important concern for strong APA scheduling, however, is how to account for the overheads induced by shifting migrations, which is addressed in the next section.

VI. Overhead Analysis

In order to assess the potential for strong APA scheduling in practice, it is necessary to provide means for overhead accounting. The overheads present in an APA scheduler are, in large parts, similar to what already occurs in standard schedulers, and can be accounted for using existing techniques (*e.g.*, see [9]). The only exception, however, are the additional shifting migrations, which are not covered by previous work.

Note that previous FP/JLFP schedulers enact migrations only upon the arrival of high-priority tasks. Thus, existing analyses for preemption-related overheads are able to charge the cost of a preemption to the higher-priority (preempting) job, which indirectly interferes with the lower-priority jobs. Under strong APA scheduling, however, tasks can also be shifted due to the arrival of lower-priority tasks, and thus the overheads cannot always be modeled as interference. Rather, they must be accounted for explicitly, as explained next.

The following lemma provides a bound on the number of migrations that can occur during the lifetime of a job of task T_k.

Lemma 3. *Under strong APA scheduling and in the absence of self-suspensions, a job J migrates at most twice for each job J' that arrives or departs while J is pending.*

Proof. Under a FP/JLFP policy, J migrates only due to the arrival or departure of some job J' of another task (regardless of priority). The claim follows since (in the absence of self-suspensions) each job arrives and departs exactly once. □

Assume a worst-case migration (or preemption) delay of Δ^{mig} time units. By Lemma 3, in a time window of length t, an upper bound $M_k(t)$ on the migration and preemption overhead incurred by some task T_k can be defined as follows:

$$M_k(t) = \sum_{T_i \in \bigcup_{l=0}^{m-1} RT_k(l)} 2 \cdot \left\lceil \frac{t}{p_i} \right\rceil \cdot \Delta^{mig}. \quad (6)$$

This additional delay affects the analysis in two cases. First, the overheads that directly affect task T_k must be added to the LP constraints that model the response time:

$$\forall \Pi_p \in \alpha_k : R_k^{ub} \leq e_k + M_k(t) + \sum_{T_i \in hp_k} X_{i,p}. \quad (7)$$

In addition, migrations that affect a task T_i of priority higher than T_k must be added to the interference $h_i^k(t)$, so that they are indirectly accounted for in the response time R_k:

$$h_i^{k\prime}(t) \triangleq h_i^k(t) + M_i(t). \quad (8)$$

The analysis can be further improved with two observations. First, tasks with singleton processor affinity can only suffer direct preemptions, but never shift. So, when analyzing such tasks, it is sufficient to consider only the arrival of higher-priority tasks executing on the same processor. In addition, when tasks with global affinity arrive, they always preempt the processor running the lowest-priority task, without affecting higher-priority tasks. Thus, we do not consider the arrival of a low-priority task with global affinity as a source of migrations.

Based on the described analysis techniques, we next report on our evaluation of strong APA scheduling.

VII. Evaluation

Recall from Sec. IV that enforcing the Strong APA Invariant allows higher system utilization. Indeed, in Sec. V, we established stricter bounds on the response time of real-time tasks to be scheduled under strong APA scheduling. To assess whether the theoretical benefits of strong APA scheduling also translate into increased utilization, we conducted schedulability experiments using the SchedCAT tool suite [1].

In the first set of experiments, we evaluated the analytical benefits of shifting using overhead-oblivious schedulability analysis, as described in Sec. V. In the second set of experiments, using the overhead accounting technique introduced in Sec. VI, we investigated how the schedulability under strong APA scheduling degrades with increasing migration overheads.

Next, we explain the experimental setup used in our evaluation, and then discuss the main trends.

A. Experimental Setup

Under APA scheduling, the affinity assignment adds a new dimension when defining task set parameters. Finding optimal affinity assignments is a non-trivial and still open problem. In addition, there is a further complication in that affinities specified by users do not necessarily improve the schedulability of the system. Since we desire to evaluate the benefits of shifting in such restricted scenarios, it was necessary to approximate user-defined affinity assignments in a reasonable manner.

Adhering to these requirements, we first randomly generated task sets using Emberson *et al.*'s method [16] with parameters $m \in \{4, 8, 16\}$, $n \in \{m + 1,\ 1.25m,\ 1.5m,\ 1.75m,\ 2m,\ 3m,\ 4m\}$, periods following a log-uniform distribution in $[10ms, 100ms]$, and priorities assigned according to the DkC heuristic [13]. For each generated task set, we then applied an affinity assignment step in order to emulate reasonable user-defined affinities, as described below.

Initial Affinity Assignment: Processor affinities were not completely randomized but rather uniformly sampled as either *partitioned*, *clustered*, or *global*, according to a probability ratio of the form $p/c/g$, respectively. For instance, the probability of a task being partitioned given a hypothetical probability ratio of $3/2/1$ would be $3/(3 + 2 + 1)$.

In order to approximate realistic use cases, two different configurations were evaluated, $1/1/6$ and $5/2/1$. The former configuration corresponds to a global-like scenario where a few cache-sensitive tasks are restricted to specific partitions or clusters, and the remaining tasks remain global. In contrast, the latter configuration is biased towards smaller affinities, *i.e.*, it resembles use cases requiring isolation or fault tolerance guarantees, in which critical tasks are statically assigned to different processors, while the remaining tasks are assigned clustered or global affinities.

After a task is assigned either a partitioned, clustered, or global configuration, it must still be assigned the exact set of processors. Although this processor selection affects the schedulability of the tasks, we preferred not to bias the experiment towards a specific allocation strategy as the user may intentionally decide to isolate or group certain tasks. We also preferred not to use random affinities since they may overload processors. Instead, we applied a simple utilization- and affinity-based heuristic.

According to the heuristic, a task T_i with a partitioned configuration is assigned to the processor Π_p that minimizes the sum $\sum u_k/|\alpha_k|$, over all tasks $T_k \in hp_i$ already assigned to Π_p. This heuristic captures the fact that high-priority tasks with larger affinities can migrate more easily, decreasing the load on the individual processors in their respective affinities. A similar heuristic was used for tasks with clustered configurations.

However, despite using the aforementioned heuristic, certain processors were still overloaded. In order to discard such clearly unschedulable assignments, we applied Baruah and Brandenburg's APA feasibility test [6].

This procedure generates the task sets we used in the evaluation, where tasks are assigned an initial processor affinity α_i. In the next step, we applied schedulability tests, as described in the following.

B. Runtime Affinity Minimization and Schedulability Tests

We consider a scenario in which a system designer has determined that certain tasks should be restricted to execute only on certain processors for application-specific reasons. However, during system integration, it is possible to *further restrict* the set of permissible processors. That is, under APA scheduling, a task T_i with a user-provided affinity α_i can be further restricted to execute on a smaller set of processors $\beta_i \subseteq \alpha_i$ without violating the user-provided scheduling restriction—there is no requirement that T_i must actually be given the option at runtime to execute on all processors in α_i *as long as schedulability is maintained*. Thus, during system integration, the user-provided affinity may be "shrunk" to obtain a simpler scheduling problem. In the following, to distinguish between the two cases, we refer to α_i as the *design-time* processor affinity of T_i and to β_i as the *runtime* processor affinity of T_i: α_i is provided by the system designer, whereas β_i is given to the OS scheduler.

Partitioning: We first evaluated task sets under partitioned scheduling, *i.e.*, by attempting to find singleton runtime processor affinities that are compliant with the user-provided design-time affinities. This partitioning-first approach reflects the observation that runtime overheads are low under partitioned scheduling and uniprocessor schedulability analysis is accurate. Therefore, there is little incentive to use an APA scheduler at runtime if partitioning suffices to meet all design-time affinity restrictions.

Finding singleton runtime processor affinities that are compliant with the user-provided design-time affinities, however, required partitioning heuristics specific for APA scheduling. Partitioning with affinity constraints is inherently limited by the initial design-time affinity assignment. Conventional bin-packing heuristics such as *First-Fit Decreasing-Utilization* are not effective. Since some tasks are assigned to single processors in the initial affinity assignment, part of the task set may already be partitioned, reducing the solution space for the bin-packing heuristics. In particular, if tasks with large affinities are partitioned first (*i.e.*, tasks whose processor affinities include most of the processors available in the system), the freedom to choose processors for the remaining tasks is reduced, affecting schedulability. Thus, we obtained improved results by sorting tasks in increasing order w.r.t. the size of their affinities.

Response-time Analysis: For the cases where partitioning failed, we applied the LP-based APA response-time analysis (described in Sec. V) on different runtime affinity assignments. Starting from the failed partitioning attempt, we first tried intermediate configurations by growing the runtime processor affinity of the tasks that did not fit during bin-packing (without violating the design-time affinity restrictions). In the cases where all these configurations were not schedulable, we applied the schedulability test on the initial design-time affinity.

Having explained the experimental setup, we next present the main results of our schedulability experiments.

C. Evaluation of Theoretical Benefits

In the first set of experiments, we assessed schedulability ignoring overheads. We generated graphs showing the fraction of schedulable task sets w.r.t. increasing system utilization (0 to m in steps of 0.05). For each point, 800 task sets were generated.

Each graph shows five curves. *PART* denotes the partitioning heuristic described previously, *i.e.*, APA scheduling where all tasks are assigned singleton runtime affinities. *RTA-Weak* and *RTA-Strong* denote the LP-based response-time analyses. The former was proposed first in [20], while the latter is described in Sec. V. To provide upper bounds on schedulability, we applied two other tests, *SIM-Weak* and *SIM-Strong*, which simulated the task sets for up to $500s$ under weak and strong APA scheduling, respectively. Since such simulations cannot consider every possible task arrival sequence, they only report task sets that are *not schedulable*. Nevertheless, they aid in estimating the gap between a scheduling algorithm and its corresponding analysis.

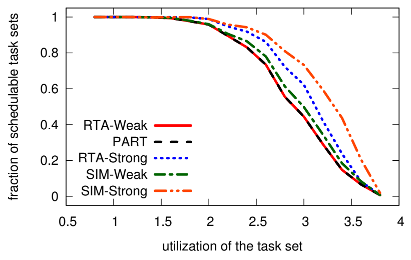
Schedulability results for 4, 8, and 16 processors, with $n \in \{1.75m, 4m\}$ and ratio $5/2/1$ are shown in Figs. 3(a)–3(f). We omit the graphs for the ratio $1/1/6$ since most task sets were partitionable and there were no differences between the curves up to very high utilization values.

Recall that a task T_k only benefits from shifting if higher-priority tasks that execute on processors in α_k can migrate to processors external to α_k, which occurs more frequently when the higher-priority tasks have large affinities. In the affinity assignments that we generated, many tasks were able to shift, allowing a significant schedulability improvement. As seen in Figs. 3(a)–3(c), RTA-Strong was able to schedule roughly 15-20% more task sets than RTA-Weak for certain utilizations.

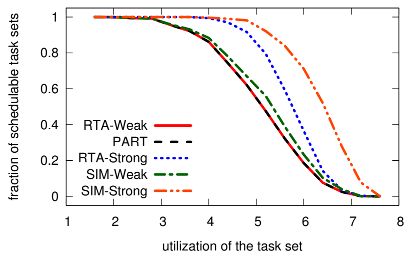
Most importantly, the fact that the *pessimistic* RTA-Strong analysis dominates the *optimistic* SIM-Weak simulation provides strong evidence demonstrating that strong APA scheduling is clearly superior in terms of schedulability.

Increasing the number of tasks reduces the difference between partitioned and APA schedulability analysis due to pessimism, as shown by comparing Figs. 3(a)–3(c) with Figs. 3(d)–3(f). However, note that SIM-Strong still suggests much better schedulability, which hints that strong APA scheduling still has a large potential that could be explored by improved analysis. Another observation is that the results for partitioned scheduling do not improve as expected for large task counts. This can be attributed to the restrictive initial affinity assignments.

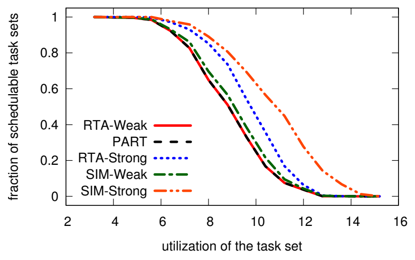
Overall, the results show that strong APA scheduling is able to achieve reasonable schedulability gains from an analytical point of view, which could be further improved by selecting better runtime affinities. In addition, all of our experiments were



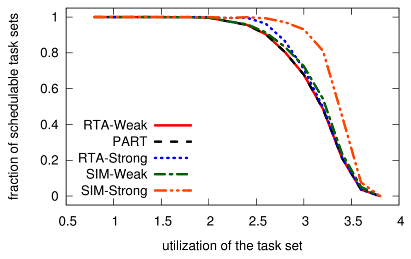
(a) $m = 4$, $n = (1.75 \cdot m) = 7$



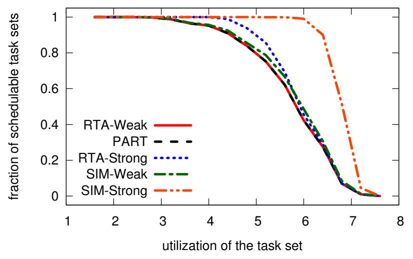
(b) $m = 8$, $n = (1.75 \cdot m) = 14$



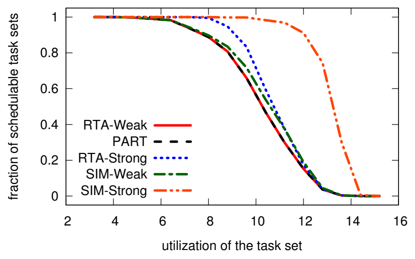
(c) $m = 16$, $n = (1.75 \cdot m) = 28$



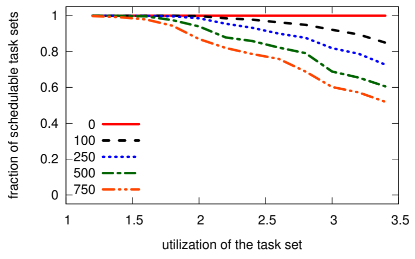
(d) $m = 4$, $n = (4 \cdot m) = 16$



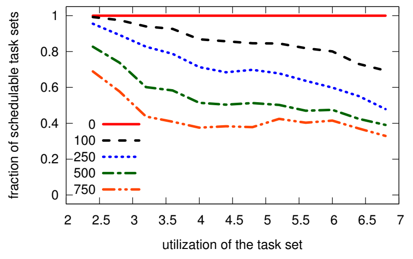
(e) $m = 8$, $n = (4 \cdot m) = 32$



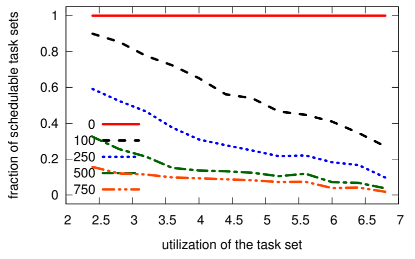
(f) $m = 16$, $n = (4 \cdot m) = 64$



(g) $m = 4$, $n = (1.50 \cdot m) = 6$



(h) $m = 8$, $n = (1.25 \cdot m) = 10$



(i) $m = 8$, $n = (1.75 \cdot m) = 14$

Fig. 3. Schedulability graphs for the partitioned/clustered/global ratio of $5/2/1$. Insets (a)-(f) do not reflect overheads. Insets (g)-(i) show results for strong APA scheduling with migration costs. The x and y axes represent the total utilization and the fraction of schedulable task sets, respectively.

limited by the initial affinity assignment, which was in part random to avoid bias towards specific application requirements.

D. *Evaluation of Shifting Overheads*

In this section, we present the results of our preliminary experiments to evaluate the practicality of strong APA scheduling, for which we used the overhead analysis from Sec. VI.

Task sets and processor affinities were generated using the same methodology as described earlier, with a small difference. Since partitioned scheduling incurs very low overheads and can always be applied without violating affinities, it is not useful as a baseline for comparison. Thus, we discarded task sets that could be partitioned and focused on only the ones that were schedulable using strong APA scheduling.

Experiment results for $m \in \{4, 8\}$ showing the fraction of task sets schedulable as the system utilization is increased from $0.3m$ to $0.9m$, and with migration costs varying from 0 to $750\mu s$, are shown in Figs. 3(g)–3(i). For small task counts and migration overheads up to $500\mu s$, a significant fraction of the task sets is deemed schedulable. Because of the large pessimism in the analysis, however, the overheads tend to grow with high task and/or processor counts. Thus, the results are not yet entirely satisfactory; however, even $500\mu s$ migration overheads can be considered quite large, and efficient implementations, if carefully optimized, may exhibit lower overheads in practice.

A full in-kernel implementation, the subject of future work, will allow measuring realistic overheads and further clarify the results. In addition, the overhead analysis could be made less pessimistic. It is clear that not all patterns of shifting can occur in real sequences of scheduling events. Quantifying possible shift patterns would improve the overhead analysis, in comparison with the current method that simply assumes that every job causes shifting twice. We thus believe that more efficient ways for implementing and analyzing APA schedulers can be found, which we leave as future work.

VIII. Related Work

Real-time APA scheduling has been addressed thrice in the literature. Our previous work focused on schedulability analysis for weak APA scheduling [19, 20], whereas Baruah and Brandenburg proposed feasibility analysis for APA scheduling with implicit deadlines [6]. In the following, we compare APA scheduling with other related scheduling problems.

Among the schedulers found in the literature, there are classes of hybrid approaches that resemble APA scheduling in the sense of explicitly controlling task migration. Since limiting *when* task migrations occur (as in restricted-migration scheduling [3, 14]) is completely orthogonal to APA scheduling, we focus on schedulers that restrict *where* a task may migrate to.

Under *clustered* scheduling [5, 11], migrations are disallowed between groups of processors, in order to decrease cross-socket memory access overheads in multicore platforms. Since such restriction can be straightforwardly encoded with affinities, APA scheduling clearly generalizes this approach [19, 20].

Semi-partitioned algorithms [4, 10, 22], while relying mostly on static task allocation schemes, still allow a small number of migratory tasks to increase schedulability. Although APA scheduling with time-varying processor affinities could model semi-partioning-like behavior, the current work on APA scheduling is still limited to static processor affinities.

Like APA scheduling, virtual cluster-based scheduling [15] also allows tasks to be assigned to overlapping physical clusters. However, Easwaran *et al.*'s work fundamentally targets an orthogonal problem of hierarchical scheduling, and also differs in that it does not fit into established scheduling APIs without breaking legacy compatibility.

IX. Conclusion

This paper, motivated by the use of arbitrary processor affinities for application-specific use cases, showed how a conventional APA scheduler can be extended to achieve higher schedulability, while maintaining API compatibility.

With the concept of task shifting, which we related to the assignment problem with seniority constraints [12], we defined how priority-based APA schedulers should be modeled in order to maximize system utilization. Though we focused on FP schedulers, the idea generalizes well to JLFP APA schedulers. In addition, apart from identifying MVMs as a tool to compute scheduling decisions, we also proposed a dynamic algorithm for implementing APA schedulers more efficiently.

To assert temporal guarantees under strong APA scheduling, we extended the recently proposed LP-based response time analysis to account for task shifting. Since overheads must be considered in practice, we also provided a preliminary method to account for shifting migrations, which can be easily adapted into the conventional overhead-aware analysis.

In our schedulability evaluation, strong APA scheduling exhibited reasonable improvements in comparison with partitioning. More importantly, despite being subject to some pessimism, strong APA schedulability analysis was shown to dominate the simulation results for weak APA scheduling. However, to confirm that our observations hold even with real-world overheads, an in-kernel implementation of the scheduler is still necessary. Due to space constraints, however, we had to relegate our ongoing implementation efforts to future work.

Finally, being a relatively new research direction, APA scheduling still has many loose ends. For instance, both the schedulability and the overhead analysis could be further improved. While the former does not yet integrate some of the more recent improvements in global schedulability analysis (*e.g.*, see [8]), the latter incurs pessimism by overcounting migrations. Nonetheless, this paper and the prior work [6, 19, 20] already show that both strong and weak APA scheduling have interesting properties that call for further exploration.

References

[1] "SchedCAT: Schedulability test collection and toolkit," project web site, http://www.mpi-sws.org/~bbb/projects/schedcat.

[2] R. Alfieri, "Apparatus and method for improved CPU affinity in a multiprocessor system," 1998, US Patent 5,745,778.

[3] J. Anderson, V. Bud, and U. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *ECRTS*, 2005.

[4] B. Bado, L. George, P. Courbin, and J. Goossens, "A semi-partitioned approach for parallel real-time scheduling," in *RTNS*, 2012.

[5] T. Baker and S. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," *Handbook of Real-Time and Embedded Systems*, 2006.

[6] S. Baruah and B. Brandenburg, "Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities," in *RTSS*, 2013.

[7] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *RTSS*, 2007.

[8] M. Bertogna and S. Baruah, "Tests for global EDF schedulability analysis," *Journal of Systems Architecture*, vol. 57, no. 5, pp. 487–497, 2011.

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

[10] 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, no. 1, pp. 3–33, 2012.

[11] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *ECRTS*, 2007.

[12] G. Caron, P. Hansen, and B. Jaumard, "The assignment problem with seniority and job priority constraints," *Operations Research*, vol. 47, no. 3, pp. 449–453, 1999.

[13] 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.

[14] F. Dorin, P. Yomsi, J. Goossens, and P. Richard, "Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms," in *RTNS*, 2010.

[15] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiprocessor scheduling," *Real-Time Systems*, vol. 43, no. 1, pp. 25–59, 2009.

[16] 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.

[17] A. Foong, J. Fung, and D. Newell, "An in-depth analysis of the impact of processor affinity on network performance," in *ICON*, 2004.

[18] A. Foong, J. Fung, D. Newell, S. Abraham, P. Irelan, and A. Lopez-Estrada, "Architectural characterization of processor affinity in network processing," in *ISPASS*, 2005.

[19] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities," in *ECRTS*, 2013.

[20] ——, "Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory," *Real-Time Systems*, vol. 50, no. 1, pp. 1–44, 2014.

[21] H.-C. Jang and H.-W. Jin, "Hoti," in *Proceedings of the 17th IEEE Symposium on High Performance Interconnects*, 2009.

[22] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *ECRTS*, 2009.

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

[24] E. Markatos and T. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," in *Supercomputing*, 1992.

[25] A. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, MIT, 1983.

[26] J. Salehi, J. Kurose, and D. Towsley, "The effectiveness of affinity-based scheduling in multiprocessor networking," in *INFOCOM*, 1996.

[27] T. Spencer and E. Mayr, "Node weighted matching," in *Automata, Languages and Programming*. Springer Berlin Heidelberg, 1984, vol. 172, pp. 454–464.

[28] A. Volgenant, "A note on the assignment problem with seniority and job priority constraints," *European Journal of Operational Research*, vol. 154, no. 1, pp. 330–335, 2004.