ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
"Antonio Ruberti"
CONSIGLIO NAZIONALE DELLE RICERCHE

V. Bonifaci, B. Brandenburg, G. D'Angelo,
A. Marchetti-Spaccamela

Multiprocessor Real-Time Scheduling with Hierarchical Processor Affinities

R. 04, 2016

Vincenzo Bonifaci – Istituto di Analisi dei Sistemi ed Informatica – CNR (`vincenzo.bonifaci@iasi.cnr.it`).

Björn Brandenburg – Max Planck Institute for Software Systems (`bbb@mpi-sws.org`).

Gianlorenzo D'Angelo – Gran Sasso Science Institute (`gianlorenzo.dangelo@gssi.infn.it`).

Alberto Marchetti-Spaccamela – Sapienza Università di Roma (`alberto@dis.uniroma1.it`).

This paper has passed an Artifact Evaluation process.
For additional details, please refer to `http://ecrts.org/artifactevaluation`.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", CNR

via dei Taurini 19, 00185 ROMA, Italy

tel. ++39-06-49937101/02
fax ++39-06-49937106
email: `iasi@iasi.cnr.it`
URL: `http://www.iasi.cnr.it`

Abstract

Many multiprocessor real-time operating systems offer the possibility to restrict the migrations of any task to a specified subset of processors by setting *affinity masks*. A notion of “strong arbitrary processor affinity scheduling” (strong APA scheduling) has been proposed; this notion avoids schedulability losses due to overly simple implementations of processor affinities.

Due to potential overheads, strong APA has not been implemented so far in a real-time operating system. We show that, in the special but highly relevant case of *hierarchical* processor affinities (HPA), strong APA scheduling can be implemented with a vastly improved runtime complexity. In particular, we present a strong HPA scheduler with a runtime complexity of $O(m)$ per task arrival and $O(\log n + m^2)$ per task departure, where m is the number of processors and n is the number of tasks, thus improving on the previous bounds of $O(m^2)$ and $O(mn)$. The improved runtime algorithms allowed us to implement support for strong hierarchical processor affinities in LITMUS$^{\mathrm{RT}}$. We benchmarked this implementation on a 24-core platform and observed non-negligible, but still viable runtime overheads.

Additionally, in the case of a bilevel affinity hierarchy and when job priorities are based on deadlines, we argue that the performance of our strong HPA scheduler, HPA-EDF, can be related to system optimality in the following way: any collection of jobs that is schedulable (under *any* policy) on m unit-speed processors subject to hierarchical affinity constraints is correctly scheduled by HPA-EDF on m processors of speed 2.415.

Key words: migration, affinity mask, arbitrary processor affinity, hierarchical structure, RTOS

AMS subject classifications: 68M07, 68W05, 90B35

1. Introduction

Most modern multiprocessor real-time operating systems offer the possibility to restrict task migration with *affinity masks*, which specify on a per-task basis on which processors a task may be scheduled. The usefulness of processor affinities in several contexts such as application performance, fault tolerance or security is well-documented [2, 21, 22, 26, 28, 32].

More recently, the problem of scheduling real-time workloads with *arbitrary processor affinities* (APAs) has been considered [5, 15, 23, 24]. To avoid schedulability losses due to overly simple implementations of such processor affinities, two notions of scheduling with arbitrary processor affinities, *weak* and *strong* APA scheduling, have been identified in prior work [15]. Commonly used schedulers, such as Linux's push-and-pull scheduler, implement only weak APA scheduling. However, it has been demonstrated that strong APA scheduling provides improved schedulability, and that it can be realized by leveraging the concept of task shifting, i.e., by allowing higher-priority tasks to be moved among processors in order to make room for lower-priority tasks that are limited by affinity constraints.

Previous work has shown that strong APA scheduling can be implemented with a runtime cost of $O(m^2)$ per task arrival and $O(mn)$ per task departure, where m is the number of processors and n is the number of tasks [15]. The second bound could be large when there many tasks. We remark that it might be difficult to improve these bounds in general, due to the combinatorial structure of the underlying matching problem.

However, we observe that in many practical scenarios affinity masks are not at all arbitrary; rather, they often follow a *hierarchical structure*, since affinity masks commonly mirror the underlying hardware topology. For example, some particular cache-sensitive, high-frequency tasks may be partitioned to a specific processor to ensure L1-cache affinity, whereas other tasks may be restricted to a subset of cores that share an L3 cache, while others yet may be assigned a global affinity mask to optimize their average-case response times.

We formalize this notion of hierarchical affinities by requiring that affinity masks follow a *laminar* set structure, that is, given any two affinity masks α and β, either α is a subset of β, or vice versa, or the two sets of processors are disjoint. This definition reflects the tree-like structure of the memory hierarchy.

Our results. We show that for such *hierarchical processor affinities* (HPAs), strong scheduling can be implemented with a vastly improved runtime complexity. In particular, we present a strong HPA scheduler with a runtime complexity of $O(m)$ per task arrival and $O(\log n + m^2)$ per task departure, where m is the number of processors and n is the number of tasks, thus improving on the previous bounds of $O(m^2)$ and $O(mn)$, respectively, whenever $n > m$ (Section 3).

Additionally, in the case of a bilevel affinity hierarchy and when job priorities are based on deadlines, we argue that the performance of our strong HPA scheduler, HPA-EDF, can be related to system optimality in the following way: any collection of jobs that is schedulable (under *any* policy) on m unit-speed processors with the given affinity constraints, is correctly scheduled by HPA-EDF on m processors of speed 2.415 (Section 4).

Finally, we experimentally validate our approach by implementing a version of our strong HPA scheduler in LITMUS$^{\mathrm{RT}}$ [1, 9, 13], a real-time extension of the Linux kernel (Section 5). To the best of our knowledge, this is the first implementation of a strong HPA scheduler in a real OS; the experiments confirm that, although the overheads are non-negligible, the proposed scheduling approach is viable in practice.

Related work. Real-time scheduling algorithms are typically classified based on the degree of migrations allowed: unrestricted migrations, no migrations, or hybrid approaches with an intermediate degree of migration. *Global scheduling* algorithms allow unrestricted migration of tasks across (possibly) all processors, while *partitioned scheduling* algorithms do not allow migration at all [18]. Proposed hybrid scheduling algorithms include *clustered scheduling* (see, for example, [6, 14]), *semi-partitioned scheduling* (see, for example, [3, 4, 12, 27]) and *restricted-migration scheduling* (see, for example, [3, 19, 30]). It is well-known that partitioning incurs lower runtime overheads, but produces schedules that may be unnecessarily constrained; global scheduling, vice versa, entails higher runtime costs that should be properly taken into account.

A simple but important observation is that APA and HPA scheduling simultaneously generalize global, clustered, and partitioned scheduling, since they allow to confine each task's migrations to a specified set of processors. An APA/HPA taskset can thus be modeled as a global, clustered, or partitioned taskset with an appropriate processor affinity assignment [5, 15, 23, 24].

Semi-partitioned scheduling relaxes partitioned scheduling by allowing a small number of jobs to migrate, thereby improving schedulability [3, 4, 12, 27]. Such tasks are called migratory, in contrast to fixed tasks that do not migrate. One difference with APA/HPA scheduling is that, in the latter models, if and when a task migrates is determined dynamically "on the fly", as under global scheduling, whereas semi-partitioned schedulers usually restrict tasks to migrate at pre-determined points in time (with regard to a job's release time) to pre-determined processors.

Clustered scheduling is another proposal that aims to alleviate limitations of partitioned and global algorithms: tasks are statically assigned to clusters of cores (like in partitioning), but are globally scheduled within each cluster [6, 14]. Clusters are disjoint and typically defined to mirror the boundaries of shared caches at a specific level in the memory hierarchy (e.g., L2 or L3). The hierarchical processor affinity model considered in this work can be interpreted as a multilevel generalization of clustered scheduling, allowing nested cluster structures.

In restricted-migration scheduling, migrations are allowed only at job boundaries [3, 19, 30]. This limits when a job may migrate, whereas APA and HPA scheduling (like global, clustered, and semi-partitioned scheduling) primarily specify where a job may migrate to. Note, however, that both global and semi-partitioned scheduling can be combined with restricted-migration scheduling, and similar approaches could also be explored in the case of APA/HPA scheduling.

Finally, if one ignores the recurrent nature of tasks, HPA scheduling is similar to a non-real-time scheduling problem in which a set of non-recurrent jobs is to be scheduled on a set of restricted machines with nested affinity structure [29]. The latter problem has recently been generalized to heterogeneous platforms [7], but to the best of our knowledge it has never been studied in the context of recurrent real-time task models.

2. System Model and Notation

We are given a set of n sporadic tasks $\tau = \{T_1, T_2, \ldots, T_n\}$ to be scheduled on a set of m processors $\pi = \{\Pi_1, \Pi_2, \ldots, \Pi_m\}$. Each task $T_i = (c_i, d_i, p_i)$ is characterized by a *worst-case execution time* c_i, a *relative deadline* d_i, and a *minimum inter-arrival time* or *period* p_i. We assume that c_i, d_i, and p_i are integers and that the tasks have *constrained* deadlines, that is, $d_i \leq p_i$. The extension of the sporadic task model considered here (proposed in [5, 15, 23]) associates with each task $T_i \in \tau$ a *processor affinity* mask $\alpha(T_i) \subseteq \pi$ that is the set of processors on which the jobs of T_i are allowed to be scheduled. We abbreviate $\alpha(T_i)$ as α_i. We assume that the family of affinity masks is *hierarchical* (or *laminar*), that is, for each $i, k = 1, \ldots, n$, either $\alpha_i \subseteq \alpha_k$ or $\alpha_k \subseteq \alpha_i$ or $\alpha_i \cap \alpha_k = \emptyset$. The *level* of an affinity mask α is the number of distinct affinity masks β such that $\beta \subseteq \alpha$. The *height* h of a task system is the maximum level among all the affinity masks of the task system. Note that $h \leq m$ since the affinity masks of a task system form a laminar family; moreover, by standard combinatorial arguments [33, Theorem 3.5], there are at most $2m$ distinct affinity masks.

Priority assignment policies used in real-time scheduling can be classified as *task-level fixed priority* (FP), *job-level fixed priority* (JLFP), or *job-level dynamic priority* (JLDP). Our model applies to both FP and JLFP scheduling. In the case of FP policies, we denote by ϕ_i the priority of T_i. In the case of JLFP policies, we denote by ϕ_i (at any time) the priority of the unique pending job of T_i (at that time); note that if T_i has no pending job, we will never consider ϕ_i.

Let $\tau(t)$ be the set of ready tasks at time t. We represent the scheduler state at time t by a bipartite graph $G(t) = (\tau(t) \cup \pi, E(t))$, where arc (T_i, Π_j) belongs to $E(t)$ iff $\Pi_j \in \alpha_i$. Hence, finding a valid allocation of tasks in $\tau(t)$ to processors π is equivalent to finding a matching $\chi(t)$ in $G(t)$.

However, not all matchings are equally desirable; in particular, one would like to maximize the number of non-idle processors while mantaining the specified priority ordering, without causing affinity violations. Note that in some cases, a processor may have to idle even though tasks are waiting. Two notions have been proposed to formalize how a correct scheduler should behave in this context by Cerqueira et al.[15].

Definition 2.1 (Weak Invariant) *At any time t, for each ready task T_b not matched by $\chi(t)$ and for each $\Pi_j \in \alpha_b$, there exists a task T_i such that $(T_i, \Pi_j) \in \chi(t)$ and $\phi_i \geq \phi_b$.*

As discussed in [15], the above requirement does not consider possible task *shiftings* that could improve schedulability without violating the affinity constraints. To take shiftings into account, a stronger definition is required based on alternating paths in the graph $G(t)$. Given a task T_b not matched to any processor by $\chi(t)$, an *alternating path* $(T_b = T_{\ell_0}, \Pi_{j_1}, T_{\ell_1}, \ldots, \Pi_{j_k}, T_{\ell_k})$, $k \geq 0$, from T_b to task T_{ℓ_k} is a path in $G(t)$ where $(T_{\ell_q}, \Pi_{j_q}) \in \chi(t)$ and $(T_{\ell_{q-1}}, \Pi_{j_q}) \in E(t)$, for each $q = 1, \ldots, k$. A processor Π_j is *reachable* from T_b according to $\chi(t)$ if there exists an alternating path from T_b to a task T_ℓ such that $\Pi_j \in \alpha_\ell$. Let $R_b(t)$ denote the set of processors reachable from task T_b in $G(t)$ with respect to the matching $\chi(t)$.

Definition 2.2 (Strong Invariant) *At any time t, for each ready task T_b not matched according to $\chi(t)$ and for each $\Pi_\ell \in R_b(t)$, there exists a task T_i such that $(T_i, \Pi_\ell) \in \chi(t)$ and $\phi_i \geq \phi_b$.*

We will need Hall's Theorem [25], a classical result on the existence of matchings in bipartite graphs.

Theorem 2.3 (Hall's Theorem) *A bipartite graph $G = (X \cup Y, E)$ has a complete matching from X to Y if and only if $|\Gamma(S)| \geq |S|$ holds for every $S \subseteq X$, where*

$$\Gamma(S) = \{y \in Y \mid (x, y) \in E \text{ for some } x \in S\}.$$

3. Scheduling with hierarchical affinities

In this section we describe an algorithm for scheduling task sets with hierarchical affinities. First we give a conceptual description of the algorithm (Section 3.1), then we prove that the assignment produced by the algorithm satisfies the strong APA invariant (Section 3.2), and finally we show how to implement the algorithm in an efficient way (Section 3.3).

3.1. Admission algorithm and feasibility

The algorithm is conceptually divided into two phases. In the first phase, we select a set τ' of tasks in $\tau(t)$ that will be scheduled at time t. Tasks in τ' are selected in such a way that there exists an assignment of tasks in τ' to processors in π that respects the affinity masks and satisfies the strong APA invariant. In the second phase we find a feasible assignment of tasks in τ' to processors in π according to their affinity masks.

The first phase of the algorithm selects the tasks in τ' in a bottom-up order, i.e., from those with the smallest affinity masks to those with the largest one. At the beginning $\tau' = \tau(t)$. For each $T_i \in \tau$, let MARK$[i]$ be a boolean variable that is `false` if and only if $T_i \in \tau(t)$ and the affinity mask α_i have not been analyzed by the algorithm. Let α_i be a minimal affinity mask that has not been analyzed by the algorithm, that is MARK$[i]$ = `true` and there is no task T_k in $\tau(t)$ such that $\alpha_k \subset \alpha_i$ and MARK$[k]$ = `false`. The algorithm iteratively applies the following procedure until all affinity masks of tasks in $\tau(t)$ have been analyzed, that is until MARK$[k]$ = `true` for all $T_k \in \tau$ (note that MARK$[k]$ = `true` for each $T_k \in \tau \setminus \tau(t)$):

1. Remove from τ' all the tasks τ_k that do not belong to the $|\alpha_i|$ highest-priority tasks among those with affinity mask $\alpha_k \subseteq \alpha_i$;

2. Set MARK$[k]$ to `true`, for each k such that $\alpha_k = \alpha_i$.

The pseudocode of the first phase is reported in Algorithm 1.

Algorithm 1: (Conceptual) Admission algorithm.

```
1  $\tau' \leftarrow \tau(t)$;
2  MARK$[i] \leftarrow$ false for each $i = 1, \ldots, n \,:\, T_i \in \tau(t)$;
3  MARK$[i] \leftarrow$ true for each $i = 1, \ldots, n \,:\, T_i \notin \tau(t)$;
4  while $\exists i : \neg$MARK$[i]$ do
5      let $i : \neg$MARK$[i]$ and $\forall k(\alpha_k \not\subset \alpha_i \vee$ MARK$[k])$;
6      $S \leftarrow \{T_k \in \tau' \,:\, \alpha_k \subseteq \alpha_i\}$;
7      sort $S$ by priority;
8      let $D$ be the $|S| - |\alpha_i|$ lowest priority tasks in $S$;
9      $\tau' \leftarrow \tau' \setminus D$;
10     MARK$[k] \leftarrow$ true, $\forall k \,:\, \alpha_k = \alpha_i$;
```

For each $i = 1, 2, \ldots, n$, let S_i be the set of tasks with an affinity mask $\alpha_k \subseteq \alpha_i$ that are in τ' at the end of the algorithm, that is $S_i = \{T_k \in \tau' \,:\, \alpha_k \subseteq \alpha_i\}$.

The next lemma shows that there exists an assignment of tasks in τ' to processors in π that satisfies their affinity masks.

Lemma 3.1. *At the end of Algorithm 1 there exists a matching in $G(t)$ of all the tasks in τ' to processors in π.*

Proof. For any set $S \subseteq \tau$ of tasks, let $\Gamma(S)$ be the set of neighbors of S in the bipartite graph $G(t)$ defined previously. We claim that

$$|\Gamma(S_i)| \geq |S_i| \text{ for every } T_i \in \tau'. \tag{1}$$

Consider the iteration relative to i. By construction $|S_i| \leq |\alpha_i|$ after line 8 during this iteration. Moreover, subsequent iterations can only decrease $|S_i|$, by removing tasks from τ'.

Now observe that, by definition of the graph $G(t)$, for any $S \subseteq \tau$, $\Gamma(S) = \cup_{T_j \in S} \Gamma(\{T_j\}) = \cup_{T_j \in S} \alpha_j$. In particular, $\Gamma(S_i) \subseteq \alpha_i$. On the other hand, when $T_i \in \tau'$, $\alpha_i \subseteq \Gamma(S_i)$ because $T_i \in S_i$. So, for $T_i \in \tau'$, $\Gamma(S_i) = \alpha_i$.

We now claim that $|\Gamma(S)| \geq |S|$ for any $S \subseteq \tau'$. To prove this, we leverage the hierarchical structure of the affinity masks. Indeed, consider any $S \subseteq \tau'$. As we already argued, $\Gamma(S) = \cup_{T_i \in S} \alpha_i$. The latter set is the union of all the affinity masks of tasks in S. We can always rewrite such a union as $\alpha_{\ell_1} \cup \ldots \cup \alpha_{\ell_q}$, where all the αs are pairwise nonintersecting; for any T_i, we call the α_{ℓ_s} containing α_i the *representative* of T_i. Thus,

$$\Gamma(S) = \alpha_{\ell_1} \cup \ldots \cup \alpha_{\ell_q} = \Gamma(S_{\ell_1}) \cup \ldots \cup \Gamma(S_{\ell_q}),$$

where again the sets on the right hand side are pairwise nonintersecting. We can now bound

$$\begin{aligned} |\Gamma(S)| &= |\Gamma(S_{\ell_1})| + \ldots + \left|\Gamma(S_{\ell_q})\right| \\ &\geq |S_{\ell_1}| + \ldots + \left|S_{\ell_q}\right| \\ &\geq \left|S_{\ell_1} \cup \ldots \cup S_{\ell_q}\right| \geq |S|, \end{aligned}$$

where the first inequality is due to (1), and the third inequality follows from the fact that the affinity mask of each task in S is covered by its representative, that is, $S \subseteq S_{\ell_1} \cup \ldots \cup S_{\ell_q}$.

Theorem 2.3, with $X = \tau'$, $Y = \pi$, guarantees the existence of a valid allocation of tasks in τ' to the processors in π. ∎

3.2. Assignment algorithm and strong APA invariant

Once the set of admitted tasks τ' has been determined, Algorithm 2 constructs the assignment guaranteed to exist by Lemma 3.1 in bottom-up order, that is, it assigns the tasks in S_i to any unused processor in α_i, assuming that, for each $\alpha_k \subset \alpha_i$, all the tasks in S_k have already been assigned.

Algorithm 2: (Conceptual) Assignment algorithm.

```
1 ASSIGN[$i$] ← false for each $i = 1, \ldots, n : T_i \in \tau'$;
2 ASSIGN[$i$] ← true for each $i = 1, \ldots, n : T_i \notin \tau'$;
3 while $\exists i : \neg$ASSIGN[$i$] do
4     let $i : \neg$ASSIGN[$i$] and $\forall k(\alpha_k \not\subset \alpha_i \vee$ ASSIGN[$k$]);
5     assign $T_i$ to any unused processor in $\alpha_i$;
6     ASSIGN[$i$] ← true;
```

We denote by $\chi(t)$ the matching produced by Algorithm 2 in $G(t)$. The set of matched tasks in $\chi(t)$ is τ'. For each task $T_i \in \tau(t) \setminus \tau'$ we denote by $R_i(t)$ the set of reachable processors from T_i according to $\chi(t)$. For each $T_i \in \tau(t) \setminus \tau'$, let α_i' be the inclusion-wise minimal affinity mask that satisfies the following properties:

1. $\alpha_i \subseteq \alpha_i'$;
2. processors in α_i' are matched with jobs T_j such that $\alpha_j \subseteq \alpha_i'$, that is, there are no edges (T_j, Π_k) in $\chi(t)$ such that $\alpha_j \not\subseteq \alpha_i'$ and $\Pi_k \in \alpha_i'$.

(In other words, α_i' is the smallest affinity mask containing α_i such that all processors in α_i' are assigned tasks that have masks contained in α_i'.)

Lemma 3.2. *For each task $T_i \in \tau(t) \setminus \tau'$, $R_i(t) = \alpha_i'$.*

Proof. We first show **(i)** that there are no alternating paths from T_i to any $\Pi_j \in \pi \setminus \alpha_i'$ and then **(ii)** that there exists an alternating path from T_i to any $\Pi_j \in \alpha_i'$. If there exists an alternating path from T_i to some $\Pi_j \in \pi \setminus \alpha_i'$, then in such a path there exists a sequence (Π_k, T_ℓ, Π_j), for some $\Pi_k \in \alpha_i'$ and $T_\ell \in \tau(t)$. By the definition of an alternating path, $\Pi_j \in \alpha_\ell$ and hence $\alpha_\ell \not\subseteq \alpha_i'$, a contradiction to Condition 2 in the definition of α_i'. This proves statement (i). To prove statement (ii), let us assume that there exists a processor $\Pi_j \in \alpha_i'$ that is not reachable from T_i. For each task T_k such that $\alpha_k = \alpha_i'$, there exists an edge (T_k, Π_j) in $G(t)$. It follows that all such tasks (if any) are not reachable from T_i, that is, for each processor $\Pi_\ell \in \alpha_i'$ and for each task T_k such that $\alpha_k = \alpha_i'$ there is no edge (T_k, Π_ℓ) in $\chi(t)$. This contradicts the assumption that α_i' was minimal, since the maximal set α_i'' such that $\alpha_i'' \subset \alpha_i'$ and $\alpha_i \subseteq \alpha_i''$ would also satisfy the Conditions 1 and 2 in the definition α_i', while being strictly contained in α_i'. ∎

Theorem 3.3. *The matching produced by Algorithm 2 satisfies the Strong Invariant.*

Proof. By Lemma 3.2, it remains to be shown that, **(i)** for each task $T_i \in \tau(t) \setminus \tau'$ and for each task T_k matched to a processor in α'_i, $\phi_k > \phi_i$, and **(ii)** for each task $T_i \in \tau(t) \setminus \tau'$ no processor in α'_i is idle.

Let us consider a sequence of q iterations of Algorithm 1 and the corresponding affinity masks $\alpha_{i_0}, \alpha_{i_1}, \ldots, \alpha_{i_{q-1}}$ considered by these iterations, where:

- α_{i_j} denotes the affinity mask considered in iteration i_j, for $j = 0, 1, \ldots, q-1$;
- i_0 is the the iteration where T_i is removed from τ' (i.e. $\alpha_{i_0} \supseteq \alpha_i$);
- $\alpha_{i_{j-1}} \subset \alpha_{i_j}$, for $j = 1, 2, \ldots, q-1$;
- for each task T_ℓ such that $\alpha_\ell = \alpha_{i_{j-1}}$, S_ℓ changes at iteration i_j (i.e., there exists a task T_r such that $\alpha_r \subseteq \alpha_{i_{j-1}}$ that was not removed from τ' in iteration i_{j-1} and that is removed in iteration i_j); and
- $\alpha_{i_{q-1}} = \alpha'_i$.

To prove statement (i), we show by induction on iteration i_j that, for each task T_k such that $\alpha_k \subseteq \alpha_{i_j}$ that is not removed from τ' at the end of iteration i_j, it holds that $\phi_k > \phi_i$.

At iteration i_0, i.e., when T_i is removed from τ', the claim holds since, by the algorithm, T_i is removed from τ' as all the $|\alpha_{i_0}|$ tasks in $\{T_k : \alpha_k \subseteq \alpha_{i_0}\} \cap \tau'$ that are not removed have a higher priority than that of T_i.

Assume that the inductive claim holds for iteration i_{j-1}. Since for each task T_ℓ such that $\alpha_\ell = \alpha_{i_{j-1}}$, S_ℓ changes at iteration i_j, there exists a task T_r such that $\alpha_r \subseteq \alpha_{i_{j-1}}$ that was not removed from τ' at iteration i_{j-1} and that is removed at iteration i_j. By induction, $\phi_r > \phi_i$ and, moreover, since T_r is removed at the end of iteration i_j, for each task T_k such that $\alpha_k \subseteq \alpha_{i_j}$ that is not removed from τ' at the end of iteration i_j, it holds that $\phi_k > \phi_r$. Therefore, $\phi_k > \phi_i$.

To prove statement (ii), we show by induction that for each α_{i_j}, no processor in α_{i_j} is idle. The basis of the induction is that when T_i is removed from τ', there are $|\alpha_i|$ tasks in τ' that can only be scheduled on processors in α_i; this follows from the fact that Algorithm 1 does not remove the $|\alpha_i|$ highest priority tasks, but removes at least one task (namely, T_i). Similarly, in subsequent steps, the algorithm does not remove the $|\alpha_{i_j}|$ highest priority tasks, but removes at least one task (namely, the task that caused S_ℓ to change). ∎

3.3. Runtime scheduling algorithm

Since the algorithm as described in the previous section has a substantial time complexity (proportional to mn or worse), in this section we discuss an efficient implementation of the same algorithm. The idea is to dynamically update the data structures that maintain the assignment of the tasks to the processors, instead of recomputing them from scratch at each task arrival or departure. To this end, we make use of *strict Fibonacci heaps* and *doubly linked lists*. A strict Fibonacci heap [11] is a heap that supports the `Insert` and `FindMax` operations in $O(1)$ computational time, and the `Delete` operation in $O(\log N)$ time, where N is the number of elements in the heap. A doubly linked list, with an auxiliary vector of pointers to each element, supports the `Insert` and `Delete` operations in $O(1)$ time, and the `FindMin` operation in $O(N)$ time [17, Chapter 10.2]. The running time of our implementation is $O(m)$ and $O(\log n + m^2)$ for task arrival and task departure, respectively. For $n > m$, this improves over the APA algorithm from [15] that requires $O(m^2)$ time for task arrival and $O(mn)$ for task departure [15].

We assume without loss of generality that for any pair of consecutive time slots t and $t+1$, the sets $\tau(t)$ and $\tau(t+1)$ differ by one task T_i, that is $\tau(t+1) \setminus \tau(t) = \{T_i\}$ in the case of task arrival and $\tau(t) \setminus \tau(t+1) = \{T_i\}$ in the case of task departure. We denote by $\tau'(t)$ the set of tasks admitted to be scheduled by Algorithm 1 at time t, and by $S_k(t)$ the sets S_k of tasks in $\tau'(t)$ with affinity mask $\beta \subseteq \alpha_k$ computed by Algorithm 1 at time t, for each $k = 1, 2, \dots, n$.

For our algorithms, we use the following data structures:

- for each affinity mask β, $s[\beta]$ is a doubly linked list that contains tasks that are scheduled on processors in β, where, for each T_i in $s[\beta]$, $\alpha_i \subseteq \beta$; and
- for each affinity mask β, $b[\beta]$ is a strict Fibonacci heap that contains tasks that are backlogged, i.e., not scheduled, where, for each T_i in $b[\beta]$, $\alpha_i = \beta$.

At time t, for each $k = 1, 2, \dots, n$, we assume that: all tasks in $S_k(t)$ are scheduled according to their affinity masks, $s[\alpha_k]$ contains all the tasks in $S_k(t)$ (i.e. $s[\alpha_k] = S_k(t)$), and $b[\alpha_k]$ contains all the tasks in $\tau(t) \setminus \tau'(t)$ with affinity mask α_k.

3.3.1. Task Arrival

As in Section 3.1, we divide the computation into two phases. In the first phase, we compute the set $\tau'(t+1)$ of tasks that are admitted to be scheduled at time $t+1$. In the second phase we assign the tasks in $\tau'(t+1)$ to processors in π according to their affinity masks.

Let us assume that a new task T_i arrives at time $t+1$, that is, $\tau(t+1) \setminus \tau(t) = \{T_i\}$. In this case, the sets of tasks admitted to be scheduled by Algorithm 1 at time t and $t+1$ differ by at most one task, that is either $\tau'(t+1) = \tau'(t) \cup \{T_i\}$, or $\tau'(t+1) = (\tau'(t) \setminus \{T_r\}) \cup \{T_i\}$, for some task T_r. To identify the potential task T_r to be removed from τ', we look for an affinity mask β such that

$$\begin{aligned} &\alpha_i \subseteq \beta, \\ &|s[\beta]| = |\beta|, \\ &\beta \text{ is minimal inclusion-wise.} \end{aligned} \tag{2}$$

If such an affinity mask exists, then $\tau'(t+1) = (\tau'(t) \setminus \{T_r\}) \cup \{T_i\}$ and β is the affinity mask containing all the processors reachable from T_i in the schedule at time t (i.e., it is equal to α'_i, see Section 3.2). In fact, all the processors in β are used and for any affinity mask γ such that $\alpha_i \subseteq \gamma$ and $\gamma \subset \beta$ there exists a task T_x, with $\gamma \subset \alpha_x$, that is scheduled in a processor in γ, while for all the tasks T_y scheduled in processors of β, it holds that $\alpha_y \subseteq \beta$. If there exists an affinity mask β satisfying condition (2), then we check whether $\phi_r < \phi_i$, where T_r is the task having lowest priority among those in $s[\beta]$. In the affirmative case, we remove T_r from $s[\gamma]$, for each γ such that $\alpha_r \subseteq \gamma$, insert T_r in $b[\alpha_r]$, and insert T_i in $s[\gamma]$, for each γ such that $\alpha_i \subseteq \gamma$, otherwise we simply insert T_i into $b[\alpha_i]$. If $|s[\beta]| < |\beta|$, for all β such that $\alpha_i \subseteq \beta$, then there are enough processors in α_i to schedule also T_i, i.e. $\tau'(t+1) = \tau'(t) \cup \{T_i\}$. In this case, T_i is added to $s[\gamma]$, for each γ such that $\alpha_i \subseteq \gamma$.

Algorithm 3 presents the pseudocode of the runtime admission algorithm. First we initialize β to α_i at line 2. Then, we look for an affinity mask β satisfying condition (2) at lines 3–5. We insert T_i in $s[\gamma]$, for each γ such that $\alpha_i \subseteq \gamma$ at lines 6–7 (note that T_i can be possibly removed from such heaps later). If $|s[\beta]| = |\beta| + 1$ after the insertion of T_i, then $|s[\beta]| = |\beta|$ before the insertion (i.e., there was no idle processor in β). In this case we must remove a task T_r. We look for T_r with the operation `FindMin`() in $s[\beta]$ at line 9. This operation returns the task with the minimum priority among those in $s[\beta]$ (including T_i) and requires $O(m)$ time. Finally, we remove T_r from $s[\gamma]$, for each γ such that $\alpha_i \subseteq \gamma$, and insert T_r in $b[\alpha_r]$ at lines 10–12. We return $\tau'(t+1)$, which is equal to $s[\pi]$ after the update process.

Theorem 3.4. *Algorithm 3 has time complexity $O(m)$.*

Proof. The loops at lines 3–5 and 6–7 perform at most $O(h)$ iterations. Since each `Insert` operation requires $O(1)$ time, each iteration requires $O(1)$ time. The loop at lines 10–11 performs at most $O(h)$ iterations that require an overall $O(h)$ time since each `Delete` operation requires $O(1)$ time. The `FindMin` operation at line 9 requires $O(m)$ time. Any other operation requires $O(1)$ time. ∎

Algorithm 3: (Runtime) Admission procedure at task arrival

```
1  Let $T_i$ be the new task;
2  $\beta \leftarrow \alpha_i$;
3  while $|s[\beta]| < |\beta| \ \wedge \ \beta \neq \pi$ do
4      Let $\gamma$ be the minimal affinity mask such that $\beta \subset \gamma$;
5      $\beta \leftarrow \gamma$;
6  foreach $\gamma \ : \ \alpha_i \subseteq \gamma$ do
7      $s[\gamma]$.Insert($T_i$);
8  if $|s[\beta]| = |\beta| + 1$ then
9      $T_r \leftarrow s[\beta]$.FindMin();
10     foreach $\gamma \ : \ \alpha_r \subseteq \gamma$ do
11         $s[\gamma]$.Delete($T_r$);
12     $b[\alpha_r]$.Insert($T_r$);
13 $\tau' \leftarrow s[\pi]$;
```

To assign the tasks in $\tau(t+1)$ to suitable processors according to their affinity masks, we use, for each affinity mask β, a linked list $L[\beta]$ that contains the unused processors in β. Each linked list supports the following operations:

- `add`(Π_i): add element Π_i to the list;
- `first`(): returns the value of the first element in the list;
- `increase`(): erases the first element of the list and shift the second element to become the first element;
- `concatenate`($L_1, L_2, \ldots, L_l$): concatenates the lists L_1, L_2, $\ldots$, L_l and outputs the concatenated list.

By using an implementation that maintains a pointer to the first and the last elements of the list, the first three operations require $O(1)$ time, while the third operation requires $O(l)$ time, where l is the number of lists to be concatenated.

We assume that each task is associated with its level and that each affinity mask α is associated with all the maximal affinity masks β such that $\beta \subset \alpha$. This information can be computed offline as it does not depend on the currently present tasks. The algorithm assigns the tasks to processors level by level, starting from tasks with level-1 affinity masks. First of all we create the list $L(\alpha)$ for each α at level-1. W.l.o.g. we can assume that the union of the affinity masks at level-1 is equal to π and therefore that the level-1 lists jointly contain all the processors in π. Note that if there are processors that are not contained in any level-1 affinity mask, we can define dummy affinity masks containing such processors, with no associated task.

Let us consider the assignment of tasks that have an affinity mask at level l, assuming that all tasks with an affinity mask of a level lower than l have been assigned to suitable processors. In particular, for each level-l affinity mask α, we assume that there exists a list $L[\alpha]$ that contains the processors in α that have not yet been assigned to any task with affinity masks of a level lower than l. For each task T_{i_k} with a level-l affinity mask, we assign T_{i_k} to $L[\alpha_{i_k}]$.`first`() and invoke $L[\alpha_{i_k}]$.`increase`(). Eventually $L[\alpha_{i_k}]$ contains all the processors in α_{i_k} that have not yet been assigned to any task with affinity masks of a level *less than or equal to l*. When all such tasks have been assigned, we create the lists $L[\alpha]$, for each level-$(l+1)$ affinity mask α, by concatenating all level-l lists $L(\beta)$ such that $\beta \subset \alpha$.

Algorithm 13 shows the pseudocode of the runtime assignment algorithm. At line 1, we create lists $L(\alpha)$ for each α at level 1. To assign the tasks to processors level by level, we sort the tasks in $\tau'(t+1)$ in order of increasing levels of their affinities (line 2). Let $T_{i_1}, T_{i_2}, \ldots, T_{i_{|\tau'|}}$ be the sorted tasks, for each $k = 1, 2, \ldots, |\tau'|$, we assign T_{i_k} to $L[\alpha_{i_k}]$.`first`() and remove from $L[\alpha_{i_k}]$ the processor to which T_{i_k} is assigned (lines 5–6). After processing the last task T_{i_k} of a level ℓ, we concatenate the lists at level ℓ to create the lists at level ℓ', where ℓ' is the level of task $T_{i_{k+1}}$ (lines 8–13). Specifically, line 8 checks whether T_{i_k} is not the last job in the list and ℓ is not the highest level h. In that case, if $\ell \neq \ell'$ (note that in this case ℓ' can be greater than or equal to $\ell + 1$), for each $l = \ell + 1, \ldots, \ell'$, the lists at level $l - 1$ are concatenated to obtain lists at level l. Eventually we obtain the lists at level ℓ'. Since the affinity masks are laminar, each affinity mask is concatenated at most once by the algorithm. Therefore, this operation requires time that is proportional to the number of distinct affinity masks in the task system.

Algorithm 4: (Runtime) Assignment algorithm

```
1  Initialize $L(\alpha)$ for each level-1 affinity mask $\alpha$;
2  Sort $\tau'$ in increasing order according to tasks' levels;
3  Let $T_{i_1}, T_{i_2}, \ldots, T_{i_{\tau'}}$ be the sorted tasks;
4  foreach $k = 1, 2, \ldots, \tau'$ do
5      Assign $T_{i_k}$ to $L[\alpha_{i_k}]$.first();
6      $L[\alpha_{i_k}]$.increase();
7      Let $\ell$ be the level of $T_{i_k}$;
8      if $k < |\tau'| \wedge \ell < h$ then
9          Let $\ell'$ be the level of $T_{i_{k+1}}$;
10         if $\ell \neq \ell'$ then
11             foreach $l = \ell + 1, \ldots, \ell'$ do
12                 foreach level-$l$ affinity mask $\alpha$ do
13                     Concatenate lists $L(\beta)$ such that $\beta \subset \alpha$ and the level of $\beta$ is $l - 1$, and assign the
                       concatenation to $L(\alpha)$;
```

Theorem 3.5. *Algorithm 13 has time complexity $O(m)$.*

Proof. Line 1 requires $O(m)$ time since it invokes m times the `add` operation. By using the counting-sort algorithm to sort the tasks in $\tau'(t+1)$, line 2 requires $O(m)$ time [17, Chapter 8.1]. Lines 8–13 concatenate the lists at level l to obtain the lists at level $l+1$. Each list corresponds to a distinct affinity mask and hence this step requires an overall time that is proportional to the number of distinct affinity masks. Since the largest affinity mask is π and $|\pi| = m$, the number of distinct affinity masks in the system is at most $2m$ (see Theorem 3.5 in [33]); therefore, this step requires overall $O(m)$ time. ∎

3.3.2. Task completion

The algorithm for task completion first removes the completed task T_i from $s[\gamma]$, for each γ such that $\alpha_i \subseteq \gamma$. Then, for each distinct affinity mask β, it invokes the algorithm for handling task arrivals by using the highest-priority task T_ℓ in $b[\beta]$ as the new task. In the case that T_ℓ is eventually scheduled, it is removed from $b[\alpha_\ell]$. The pseudocode of the algorithm is given in Algorithm 5. Note that line 12 of Algorithm 3 must be modified in order to check whether task T_r is already present in $b[\alpha_r]$; this can be done in $O(1)$ time.

Algorithm 5: (Runtime) Scheduling algorithm for task completion

```
1 Let $T_i$ be the completed task;
2 foreach $\gamma$ : $\alpha_i \subseteq \gamma$ do
3     $s[\gamma]$.Delete($T_i$);
4 foreach distinct affinity mask $\beta$ do
5     $T_\ell \leftarrow b[\beta]$.FindMax();
6     Run task arrival procedure by using $T_\ell$ as new task;
7 Let $T_\ell$ be the scheduled task, if any;
8 $b[\alpha_\ell]$.Delete($T_\ell$);
```

Theorem 3.6. *Algorithm 5 has time complexity $O(\log n + m^2)$.*

Proof. Lines 2–3 require $O(h) = O(m)$ time. Lines 5 and 6 require $O(1)$ and $O(m)$ time, respectively. Since they are invoked $O(m)$ times (as the number of distinct affinity masks is $O(m)$), they require an overall $O(m^2)$ time. Line 8 requires $O(\log n)$ time. ∎

4. Bilevel and clustered scheduling

In this section, we consider a restricted scenario in which the hierarchy of affinity masks has only two levels. Namely, we assume that the task set τ is partitioned into $M+1$ sets $\tau_0, \tau_1, \ldots, \tau_M$ and the set of processors is partitioned into M sets $\pi_1, \pi_2, \ldots, \pi_M$ such that:

- $\tau = \tau_0 \cup \tau_1 \cup \ldots \cup \tau_M$ and $\tau_i \cap \tau_j = \emptyset$, for each $i \neq j$,
- $\pi = \pi_1 \cup \pi_2 \cup \ldots \cup \pi_M$ and $\pi_i \cap \pi_j = \emptyset$, for each $i \neq j$,
- for each $T_i \in \tau_0$, $\alpha_i = \pi$,
- for each $i = 1, 2, \ldots, M$ and $T_k \in \tau_i$, $\alpha_k = \pi_i$.

In other words, the affinity mask of each task T_k is:

$$\alpha_k = \begin{cases} \pi_i & \text{if } T_k \in \tau_i, \text{ for some } i > 0, \\ \pi & \text{if } T_k \in \tau_0. \end{cases}$$

Table 1 illustrates the considered affinity masks, where an entry in row i and column k of the table is 1 if tasks in τ_i can be scheduled on processors in π_k, and 0 otherwise. This scenario models the following practically relevant cases:

Table 1: Affinity masks for bilevel or clustered scheduling

	π_1	π_2	$\cdots$	π_M
τ_0	1	1	$\cdots$	1
τ_1	1	0	$\cdots$	0
τ_2	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
τ_M	0	0	$\cdots$	1

- *bilevel scheduling*, in which each task is either globally scheduled, or assigned to a specific processor, or
- *clustered scheduling*, in which each task is either globally scheduled, or assigned to a cluster of processors.

For each $i = 1, 2, \ldots, M$, we denote $|\pi_i|$ as m_i. Moreover, we define $m_{\max} = \max_{i=1}^{M}\{m_i\}$ and $m_0 = m$. We denote by J any possible collection of jobs generated by τ. For each job $j = (R_j, C_j, D_j)$ in J, we let R_j, C_j, and D_j denote the release time, the execution time requirement, and the *absolute* deadline of j, respectively. If T_i is the task that generated job j, then $C_j = c_i$ and $D_j = R_j + d_i$. Given a time instant t, a job is *available* at time t if it has been released before or at t and has not signaled its completion. Since we are considering constrained deadline tasks, at any time t, there exists at most one available job for each task.

We will analyze the previously described algorithm in the special case where each job's priority is given by its absolute deadline; we therefore call it *Hierarchical Processor Affinities – Earliest Deadline First* (*HPA-EDF*).

4.1. Idealized schedule

Given a job collection J generated by τ that is feasible on m unit speed processors under affinity mask constraints α, the following idealized algorithm, A_∞, is able to schedule J on a platform of infinitely many speed-$(2 - 1/m_{\max})$ processors $\pi \cup \{\Pi_{m+1}, \Pi_{m+2}, \ldots\}$, where the additional processors can only schedule tasks in τ_0:

1. for each $i = 1, 2, \ldots, M$, schedule the jobs of tasks in τ_i on processors in π_i by using (global) EDF; and
2. allocate one processor Π_k, $k > m$, to each job of tasks in τ_0 and schedule all such jobs as early as possible.

Let us denote by S_∞ the corresponding idealized schedule. To simplify the notation, we denote $(2 - 1/m_{\max})$ as s'.

Lemma 4.1. *At any point in time, at least as much total work has been processed in S_∞ as in any schedule which is feasible upon platform π with unit-speed processors and subject to the affinity mask constraints α.*

Proof. By construction, at any point in time, S_∞ processed at least as much of each job generated by tasks in τ_0 as any feasible schedule. If we assume that, for some $i \in \{1, 2, \ldots, M\}$, schedule S_∞ with speed $s' = 2 - 1/m_{\max} \geq 2 - 1/m_i$ provides less service to jobs of τ_i than some feasible schedule, then we obtain a contradiction with the fact that any global work-conserving schedule on m_i speed-$(2 - 1/m_i)$ processors processes at least as much work as any unit-speed schedule on m_i processors [31, Lemma 2.6]. ∎

4.2. Speedup bound for HPA-EDF

We analyze HPA-EDF on m processors of speed s, for a suitable $s \geq s'$ to be determined later. By considering sufficiently small time slots, an equivalent description of the algorithm repeats the following procedure at each time slot:

1. consider the list of available jobs;
2. for each $i = 1, 2, \ldots, M$, remove from the list all jobs of tasks in τ_i but the m_i with the earliest deadlines;
3. sort the remaining jobs in order of non-decreasing absolute deadlines;
4. consider the first m jobs in the obtained list, and
   (a) for each $i = 1, 2, \ldots, M$, schedule the job of a task in τ_i (if any) onto an empty processor in π_i;
   (b) schedule the remaining jobs (of τ_0) onto the empty processors.

Lemma 4.2. *Consider a collection J of jobs generated by τ and let $s \geq s'$. Then at least one of the following holds:*

1. *All jobs in J are completed within their deadline under HPA-EDF on m processors of speed s, or*
2. *J is not feasible under the speed-s' schedule S_∞, or*
3. *there exists an interval I such that any feasible schedule for J must finish more than $\left(sm - 2(m-1)\frac{ss'}{s+s'}\right)|I|$ units of work within I.*

Proof. Suppose that the first two conditions do not hold, that is, J is feasible under the idealized schedule S_∞ with speed-s' processors, but under HPA-EDF with speed s there is a job $j = (R_j, C_j, D_j)$ that fails to meet its deadline. We aim to show that the third condition must then be satisfied. Let us assume that j has been generated by a task T in τ_f, $f \geq 0$. We assume w.l.o.g. that J is minimal, in the sense that there are no jobs with a deadline greater than D_j.

Let us define t^* as the latest time instant such that:

- $t^* \leq R_j$, and
- at time t^*, for any available job, HPA-EDF with speed s has processed at least as much of that job as A_∞ with speed s'.

Time t^* exists because $t^* = 0$ satisfies the above conditions. Let $I = [t^*, D_j]$. We partition I into the following intervals:

- $X_k \subseteq [t^*, R_j]$: these are the intervals that occur before R_j such that all the processors are busy under HPA-EDF scheduling;
- $Y_k \subseteq [t^*, R_j]$: these are the intervals that occur before R_j such that at least one processor is empty under HPA-EDF scheduling;
- $Z_k \subseteq [R_j, D_j]$: if $f > 0$, these are the intervals that occur after R_j such that, under HPA-EDF scheduling, at least one processor in π_f is busy only with jobs from τ_f for the entire interval and at least one processor in π is empty; otherwise, they are the intervals that occur after R_j such that at least one processor is empty under HPA-EDF scheduling; and
- $W_k \subseteq [R_j, D_j]$: if $f > 0$, these are the intervals that occur after R_j such that all the processors are busy under HPA-EDF scheduling.

Note that, if $f > 0$, then the union of intervals Z_k and W_k is equal to $[R_j, D_j]$. In fact, if at some time instant all processors in π_f schedule a job in τ_0, then all processors in π must be busy, otherwise a job in τ_0 scheduled in π_f can be moved to an empty processor.

Fact 4.3. *During each interval Y_k, there are no available jobs in τ_0 that are not scheduled.*

Proof. It is enough to observe that any available job in τ_0 can be scheduled on the processor that is empty in Y_k. ∎

It follows that, if $t^* < R_j$, the first interval after t^* is of type X_k. Hence the interval $[t^*, R_j]$ is composed of a sequence of intervals $X_0, Y_0, X_1, Y_1, \ldots$

Fact 4.4. *For each interval Y_k, there exists a processor which is busy for the entire interval Y_k under HPA-EDF scheduling.*

Proof. By contradiction, let us suppose that for each processor there exists an interval within Y_k in which it is empty. For each $i = 1, 2, \ldots, M$, let us denote by $t_{k,i}$ the latest starting time of an interval within Y_k such that a processor in π_i is empty. At time $t_{k,i}$, all the jobs in τ_i released before Y_k have been processed by HPA-EDF. Moreover, for jobs in τ_i released within Y_k, HPA-EDF and A_∞ use the same priority ordering. Therefore, at time $t_{k,i}$, HPA-EDF has processed at least as many jobs in τ_i as A_∞. Similarly, at time $\hat{t} = \max_i t_{k,i}$ all the jobs in τ_0 released before Y_k have been processed by HPA-EDF and, by Fact 4.3, all the jobs in τ_0 released within Y_k are processed by HPA-EDF. Hence, at time $\hat{t}$ HPA-EDF has processed at least as many jobs in τ_0 as A_∞. It follows that, at time $\hat{t}$, for each available job, HPA-EDF has processed at least as much of that job as A_∞, a contradiction to the definition of t^*. ∎

We denote by X, Y, W, and Z the total length of intervals X_k, Y_k, W_k, and Z_k, respectively; we also group the sets X_k and Y_k into sequences $\bar{X}_\ell$ and $\bar{Y}_\ell$ as follows.

1. Let $\bar{Y}_0$ be the maximal sequence of consecutive intervals $\{Y_0, Y_1, \ldots, Y_x\}$ such that the same processor is busy in Y_k and in Y_{k+1} for $k = 0, 1, \ldots, x-1$ and let $\bar{X}_0$ be the corresponding sequence $\{X_0, X_1, \ldots, X_x\}$.
2. We remove $\bar{X}_0$ and $\bar{Y}_0$ from $\{X_k\}$ and $\{Y_k\}$, respectively, and define $\bar{X}_\ell$ and $\bar{Y}_\ell$, $\ell > 0$, by repeating the above procedure until $\{X_k\}$ and $\{Y_k\}$ become empty.

Let us denote by $|\bar{X}_\ell|$ and $|\bar{Y}_\ell|$ the cumulative length of intervals in $\bar{X}_\ell$ and $\bar{Y}_\ell$, respectively.

Fact 4.5. $s|\bar{Y}_\ell| \leq s'\left(|\bar{X}_\ell| + |\bar{Y}_\ell|\right)$.

Proof. Let us denote by $a < m$ the number of processors in π which are busy in the entire sequence of intervals $\bar{Y}_\ell$ under HPA-EDF scheduling, and let us denote by c the cumulative execution requirement of the jobs scheduled by HPA-EDF in those a processors during $\bar{Y}_\ell$. By Fact 4.4, it follows that $sa|\bar{Y}_\ell| \leq c$. Since HPA-EDF processes a jobs in parallel among those that contribute to c, and since in the intervals $\bar{Y}_\ell$ there is always an empty processor, A_∞ can process at most a such jobs in parallel. In fact, the jobs in τ_0 that contribute to c could be scheduled in a processor that is empty in $\bar{Y}_\ell$ by HPA-EDF, while jobs in τ_i, $i > 0$, follow the same priorities in A_∞ and HPA-EDF. Therefore, for jobs in c, A_∞ can have processed at most $as'|\bar{X}_\ell|$ amount of work during $\bar{X}_\ell$. Therefore, for any fixed $\delta > 0$, the maximum amount of work of c that A_∞ can execute in $\bar{X}_\ell$ and after δ amount of time in $\bar{Y}_\ell$ is $as'\left(|\bar{X}_\ell| + \delta\right)$. Moreover, after δ amount of time in $\bar{Y}_\ell$, HPA-EDF can execute at least $sa\delta$ work of c. Hence, if δ is such that $as'\left(|\bar{X}_\ell| + \delta\right) = sa\delta$ (i.e. $\delta = \frac{s'}{s-s'}|\bar{X}_\ell|$), then after δ amount of time in $\bar{Y}_\ell$ HPA-EDF has processed at least as much of c as A_∞. It follows that $c \leq as'\left(|\bar{X}_\ell| + \delta\right)$, as otherwise the definition of t^* is contradicted. Therefore, $sa|\bar{Y}_\ell| \leq as'\left(|\bar{X}_\ell| + \delta\right)$, which implies the statement by plugging in the above value of δ. ∎

By summing over all ℓ, we obtain $sY \le s'(X+Y)$.

Let $a_f \in \{1, 2, \ldots, m_f\}$ be the number of processors in π_f that are busy in the entire sequence of intervals $\{Z_k\}$ only with jobs of τ_f.

Fact 4.6. *If $a_f < m_f$, then $sZ < s'(W+Z)$.*

Proof. Since in each Z_k there exists an empty processor in π, then, in the case that $a < m_f$, there exists an empty processor in π_f in each Z_k. Therefore, the failing job j is executed during the entire sequence of intervals $\{Z_k\}$. Moreover, job j does not meet its deadline, so $sZ < C_j$. On the other hand, since A_∞ correctly schedules j and it can schedule j for at most $s'(W+Z)$ amount of time, one has $C_j \le s'(W+Z)$. It follows that $sZ < s'(W+Z)$. ∎

Fact 4.7. *If $a_f = m_f$, then $sZ < s'(|I| - Y)$.*

Proof. Let c be the cumulative execution requirement of the jobs scheduled by HPA-EDF in π_f during $\{Z_k\}$ plus job j (if it is not scheduled during $\{Z_k\}$), minus the amount of work processed by HPA-EDF for these jobs before R_j. During the sequence of intervals $\{Z_k\}$, HPA-EDF schedules sm_fZ amount of work of jobs of c and, since j fails, $sm_fZ < c$. On the other hand, since A_∞ correctly schedules all the jobs of c and it can schedule them for at most $m_fs'(|I| - Y)$ amount of time, we have $c \le m_fs'(|I| - Y)$. It follows that $sZ < s'(|I| - Y)$. ∎

By the definitions of X and W, all processors are busy in such intervals, and hence HPA-EDF executes $sm(X+W)$ units of work. By Fact 4.4, for each interval Y_k, there exists a processor which is busy for the entire interval Y_k under HPA-EDF scheduling and then HPA-EDF executes at least sY units of work. Finally, for each interval Z_k, there exists at least one processor in π_f that is busy for the entire interval and then, HPA-EDF executes at least sZ units of work. Overall, in the interval I, HPA-EDF executes at least $sm(X+W) + s(Y+Z)$ units of work. The following cases may arise.

1. If $a_f < m_f$, then by Facts 4.5 and 4.6 it follows that:

$$\begin{aligned}
&sm(X+W) + s(Y+Z)\\
&= sm(|I| - Y - Z) + sY + sZ\\
&= sm|I| - sY(m-1) - sZ(m-1)\\
&> sm|I| - s'(X+Y)(m-1) - s'(W+Z)(m-1)\\
&= (sm - s'(m-1))\,|I|.
\end{aligned}$$

2. If $a_f = m_f$, let $b \in [0,1)$, then we have two subcases.

   (a) If $X + W \le b|I|$, Facts 4.5 and 4.7 imply that:

$$\begin{aligned}
&sm(X+W) + s(Y+Z)\\
&= sm|I| - (m-1)(sY + sZ)\\
&> sm|I| - s'(m-1)(X + Y + |I| - Y)\\
&\ge sm|I| - s'(m-1)(b+1)|I|.
\end{aligned}$$

   (b) If $X + W > b|I|$, then

$$\begin{aligned}
&sm(X+W) + s(Y+Z)\\
&= sm(X+W) + s(|I| - X - W)\\
&> sb(m-1)|I| + s|I|.
\end{aligned}$$

The first obtained lower bound decreases with b, while the second one increases with b. It follows that, for the value of b for which they are equal, we obtain the maximum lower bound for any value of b. The value of b for which the two lower bounds are equal is $b = \frac{s-s'}{s+s'}$ and the obtained lower bound is $\left(sm - 2(m-1)\frac{ss'}{s+s'}\right)|I|$.

Since $s \geq s'$, the lower bound given in case 1 is always greater than that given in case 2. This proves the statement of the lemma. In fact, since HPA-EDF executes more than $\left(sm - 2(m-1)\frac{ss'}{s+s'}\right)|I|$ units of work, by the definition of I, A_∞ with speed s' executes at least the same amount of work during I. By Lemma 4.1, any feasible schedule of J cannot process more than A_∞ with speed s' before I and hence it must execute more than $\left(sm - 2(m-1)\frac{ss'}{s+s'}\right)|I|$ units of work within I. ∎

Theorem 4.8. *Any collection of jobs J generated by τ that is feasible on m unit-speed processors under affinity mask constraints α is schedulable by HPA-EDF on m processors with speed $s < 3.562$, and speed $s < 2.415$ if $m_{\max} = 1$.*

Proof. Since J is feasible on m unit speed processors under affinity mask constraints α, then at any interval I a feasible schedule can execute at most $m|I|$ units of work. If $s = \frac{s'+1}{2} - \frac{s'}{m} + \frac{\sqrt{(2s'-m(s'+1))^2+4s'm^2}}{2m}$, then $\left(sm - 2(m-1)\frac{ss'}{s+s'}\right)|I| = m|I|$. By Lemma 4.2, it follows that HPA-EDF correctly schedules J with speed s.

Since $s' \leq 2$, then $s \leq \frac{3}{2} - \frac{2}{m} + \frac{\sqrt{(4-3m)^2+8m^2}}{2m} < 3.562$; if $m_{\max} = 1$, i.e. $|\pi_i| = 1$ for each $i > 0$, then $s' = 1$ and $s = 1 - \frac{1}{m} + \frac{\sqrt{2m^2-2m+1}}{m} \leq 1 + \sqrt{2} - \frac{1}{m} < 2.415$. ∎

5. Implementation and Evaluation

From the point of view of an RTOS developer, it is not obvious that the runtime algorithm presented in Section 3.3 lends itself to implementation in a real OS. To investigate the algorithm's viability in practice and to explore implementation choices, we implemented a prototype in LITMUS$^{\mathrm{RT}}$ [1, 9, 13], a real-time extension of the Linux kernel, and conducted overhead measurements on a 24-core Intel Xeon platform.

5.1. Maintaining processor locality

One needed practical tweak pertains to Algorithm 13, which maps scheduled tasks to processors without any consideration for *processor locality*: in line 5 of Algorithm 13, each task is assigned simply to the first available processor, irrespective of where the task was scheduled previously. As a result, tasks may incur superfluous migrations. In practice, task migrations are costly, both due to the required OS and processor state updates (runqueue management, context switches, etc.) and due to the potential loss of cache affinity. Migrations should thus be avoided to the extent possible.

Fortunately, Algorithm 13 can be augmented to re-assign tasks to the last-used processor, provided this is possible without violating the strong APA invariant. Suppose we maintain for each processor a link to the list element used to enqueue it in the list $L[\cdot]$ (lines 1 and 13 in Algorithm 13), and for each task a link to the last-used processor. It is then possible to change line 5 of Algorithm 13 as follows: first check whether T_{i_k}'s last-used processor is still available (i.e., enqueued in $L[\alpha_{i_k}]$), and if so, remove the processor from the list and re-assign T_{i_k} to it. Assuming $L[\cdot]$ is realized as a doubly-linked list, this additional check and re-assignment can be carried out in $O(1)$ time, thus preserving the overall complexity.

However, even with this modification, Algorithm 13 is still oblivious as to *when* a task used a processor last, which can be problematic if multiple tasks arrive that previously executed on the same processor (at different times). We hence instead implemented Algorithm 6, which favors later-scheduled tasks.

Algorithm 6: Locality-aware assignment algorithm

```
1  Let $available$ denote the set of all processors;
2  Let $reserved$ denote an initially empty set of processors;
3  Let $pref$ denote an array of task pointers of length $m$;
4  Sort $\tau'$ in increasing order according to tasks' levels;
5  Let $T_{i_1}, T_{i_2}, \ldots, T_{i_{|\tau'|}}$ be the sorted tasks;
6  foreach $k = 1, 2, \ldots, |\tau'|$ do
7      Let $T_x = pref[T_{i_k}.\texttt{last\_cpu}]$;
8      if $T_x = \texttt{NIL} \lor T_x.\texttt{last\_t} < T_{i_k}.\texttt{last\_t}$ then
9          $pref[T_{i_k}.\texttt{last\_cpu}] \leftarrow T_{i_k}$;
10         add $T_{i_k}.\texttt{last\_cpu}$ to $reserved$;
11 foreach $k = 1, 2, \ldots, |\tau'|$ do
12     Let $valid = available \cap \alpha_{i_k}$;
13     if $T_{i_k}.\texttt{last\_cpu} \in valid \land pref[T_{i_k}.\texttt{last\_cpu}] = T_{i_k}$ then
14         re-assign $T_{i_k}$ to processor $T_{i_k}.\texttt{last\_cpu}$ and remove $T_{i_k}.\texttt{last\_cpu}$ from $available$;
15     else
16         if $valid \setminus reserved \neq \emptyset$ then
17             assign $T_{i_k}$ to any processor in $valid \setminus reserved$ and remove the processor from $available$;
18         else
19             assign $T_{i_k}$ to any processor in $valid$ and remove the processor from $available$;
```

Algorithm 6 first initializes an array of task pointers to record each task's preferred processor (lines 6–10), resolving conflicts based on where and when a task was last scheduled (denoted as T_i.`last_cpu` and T_i.`last_t`, respectively). In the second step, each task is assigned to its preferred processor (lines 13–14), unless the processor was claimed by a more-recently scheduled task. If a task's locality cannot be maintained, the algorithm first attempts to assign it to an unclaimed processor (lines 16–17), to avoid interfering with the preferences of other tasks. Finally, if this is not possible, any remaining unassigned processor in α_{i_k} may be used (lines 18–19).

Algorithm 6 uses sets instead of lists to keep track of both unallocated and reserved processors. While this choice increases the algorithm's time complexity, it can be implemented more efficiently in practice because the number of processors is typically a small constant, which allows sets of processors to be represented as word-sized bitmaps that support (effectively) constant-time operations.

5.2. Implementation in LITMUS$^{\text{RT}}$

We implemented a prototype fixed-priority scheduler based on Algorithms 3, 5, and 6 in LITMUS$^{\text{RT}}$ version 2015.1 [1, 9, 13], which in turn is based on Linux 4.1.

To allow the scheduler to scale to large multicore platforms, we adopted a design based on message passing [16] in which the system is split into a single *dedicated system management* (or *service*) *processor* (DSP), which handles all interrupts and makes all scheduling decisions, and the remaining *application processors* (AP), which actually execute the real-time workload.

The DSP is notified of job arrivals by local interrupts and of job departures by APs via an efficient wait-free message queue. In response, the DSP makes a scheduling decision (according to Algorithms 3, 5, and 6) and assigns each AP a job to execute.

Each AP runs a simple, policy-free dispatcher that simply context-switches to the currently assigned job (if any) or executes non-real-time background work (if no job is assigned). Whenever the DSP changes an AP's assignment, it sends an *inter-processor interrupt* (IPI) to invoke the AP's dispatcher.

Cerqueira et al. [16] pioneered this scheduler design for global schedulers and showed it to scale much better (in terms of worst-case overheads) than Linux's native push-and-pull scheduler (which enforces the weak APA invariant); we hence use Cerqueira et al.'s global scheduler as our baseline.

While we refer to Cerqueira et al. [16] for an in-depth discussion of the approach's advantages and disadvantages, two aspects are of particular relevance: since only the DSP makes scheduling decisions, there is no need to explicitly synchronize access to scheduler data structures (i.e., the per-affinity lists of scheduled tasks and queues of backlogged jobs), and all scheduler data structures remain cache-local to the DSP.

Finally, instead of a strict Fibonacci heap as assumed in Algorithms 3 and 5, we used an array of linked lists (one per priority level) with an associated bitmap to indicate non-empty queues, which is the standard approach for implementing fixed-priority schedulers. Given that RTOSs typically implement only a limited number of distinct priority levels (e.g., 512 in LITMUS$^{\mathrm{RT}}$), this enables (effectively) $O(1)$ queue operations with much lower constant factors than strict Fibonacci heaps.

To the best of our knowledge, this is the first implementation of a strong APA scheduler in a real OS.

5.3. Experimental Evaluation

While some extra overhead can be expected—compared to the global baseline, an APA scheduler must consider additional constraints at runtime, which does not come for free—it is difficult to anticipate the magnitude of such additional costs, which could range from negligible to crippling. To investigate the system's overheads in a practical setting, we ran the baseline and the proposed schedulers on an Intel Xeon E7-8857 platform using two sockets with twelve cores each. Each core has private L1 and L2 caches (32 KiB and 256 KiB, respectively), and shares an L3 cache (30 MiB) local to its socket.

We generated 100 task sets with Emberson et al.'s task set generator [20], each with a total utilization of either 75% or 85%, periods in the range $[1ms, 1000ms]$ drawn from a log-uniform distribution, and a number of tasks ranging from $2m = 48$ to $10m = 240$ in steps of $2m$. We considered three levels of affinities: *global* (i.e., all APs), *socket-local* (i.e., all APs sharing an L3 cache), or *partitioned* (i.e., just a single AP, which ensures L2 affinity). For each task, we selected uniformly at random a level (i.e., global, socket-local, or partitioned), and in the latter two cases then selected uniformly at random a socket or partition, respectively. Each task set was guaranteed to be feasible [5], and each c_i parameter was ensured to exceed $500\mu s$ to filter impractically small parameters. Finally, we assigned all tasks rate-monotonic priorities.

Each task set was executed under both schedulers for 60 seconds each. Overheads were collected with the *Feather-Trace* instrumentation toolkit [10] included in LITMUS$^{\mathrm{RT}}$. In total, we recorded more than 34 GiB of trace data containing more than 700,000,000 valid overhead samples.

We focus here on the high-level scheduling overheads incurred on the DSP (i.e., Algorithms 3, 5, and 6) and the low-level dispatcher overheads incurred on the APs (i.e., dispatcher invocation and context-switching costs), as these two overhead sources exhibit the most telling differences, and omit a discussion of other system overheads due to space constraints. The relevant observed overhead data is summarized in Fig. 1.

Because the memory hierarchy and the general design of commodity x86 platforms is decidedly not real-time friendly, the maximum overheads are affected by spurious outliers that exceed average and median overheads by a factor of 20x to 50x. Since such rare, but extreme outliers affect both the baseline and the HPA scheduler, they obscure a meaningful comparison.

Hence, we instead report 99.9th percentile overheads, which are not obscured by outliers, and which we anyway consider to be more relevant for the "firm" real-time systems that can be expected to be deployed on commodity

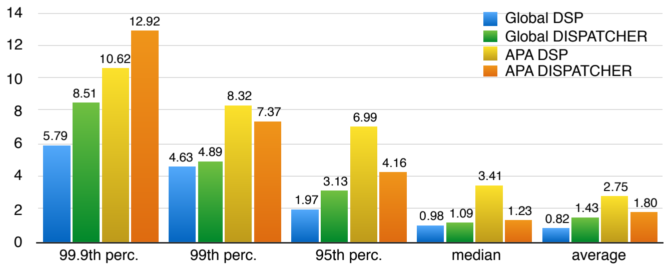

Figure 1: Measured DSP and AP dispatcher overheads (in microseconds) under the global baseline scheduler and the proposed HPA scheduler.

multicore platforms in the foreseeable future. For additional context, we also report 99th percentile, 95th percentile, median, and average overheads, which may be more relevant to soft real-time systems.

As expected, the results show the APA scheduler to incur higher overheads. In particular, the 99.9th percentile DSP overhead increased from $5.79\mu s$ to $10.62\mu s$, and this increase is reflected also in the median and average DSP overhead. However, while this is a substantial increase, it is important to note that these overheads are still within the range of a few microseconds, which is likely acceptable for tasks with a period in the range of a few to a few hundred milliseconds.

Interestingly, our data indicates that APs also experience increased dispatcher overheads, which may be surprising at first given that, under *both* schedulers, the AP dispatcher simply enacts the DSP's assignments. We attribute this increase in costs to a difference in how tasks migrate: whereas under global scheduling tasks migrate typically only after having been backlogged for some time, strong APA scheduling sometimes requires shifting migrations [23], where a task that is scheduled on one processor must continue its execution immediately on another processor. Such migrations are more costly as they require careful coordination among the involved processors.

Overall, our prototype implementation shows the proposed scheduler to be practical: while overheads are indeed higher, they remain in a feasible range. Whether or not the capability to impose APAs is worth the extra costs is ultimately a design tradeoff that is best assessed in an application-specific context.

6. Conclusion

We have studied hierarchical (i.e., laminar) affinity masks in the context of strong APA scheduling. For this important special case, which arises naturally in practice if affinity masks mirror a system's hardware topology, we have devised a new scheduler that is substantially more efficient (whenever $n > m$) than the best-known general-case strong APA scheduler [15]: our scheduler improves the per-arrival cost from $O(m^2)$ to $O(m)$, and the per-departure cost from $O(mn)$ to $O(\log n + m^2)$.

In combination with EDF and assuming bilevel or clustered affinities, we have related our scheduler to an optimal system in the following way: any collection of jobs that is schedulable (under any policy) on m unit-speed processors subject to hierarchical affinity constraints is correctly scheduled by our scheduler on m processors of speed 2.415.

Finally, we have shown the proposed scheduler to be practically viable with a fixed-priority prototype implementation in an actual RTOS, namely LITMUS$^{\mathrm{RT}}$, and overhead measurements on a 24-core Intel Xeon platform.

Acknowledgments

Work partially supported by project C26A15TXCF of Sapienza University of Rome. The authors also acknowledge Schloss Dagstuhl Seminar 16081 “Scheduling”.

References

[1] LITMUS$^{\mathrm{RT}}$: The Linux testbed for multiprocessor scheduling in real-time systems. `http://www.litmus-rt.org`.

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

[3] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *ECRTS*, pages 199–208, 2005.

[4] B. Bado, L. George, P. Courbin, and J. Goossens. A semi-partitioned approach for parallel real-time scheduling. In *RTNS*, pages 151–160, 2012.

[5] S. Baruah and B. B. Brandenburg. Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In *RTSS*, pages 160–169, 2013.

[6] S. K. Baruah and T. P. Baker. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. Chapman and Hall, 2007.

[7] V. Bonifaci, G. D’Angelo, and A. Marchetti-Spaccamela. Hierarchical and semi-partitioned machine scheduling. Technical Report 15-03, IASI-CNR, 2015.

[8] V. Bonifaci, B. Brandenburg, G. D’Angelo, and A. Marchetti-Spaccamela. Multiprocessor real-time scheduling with hierarchical processor affinities. Technical Report 16-04, IASI-CNR, 2016.

[9] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, UNC Chapel Hill, 2011.

[10] B. Brandenburg and J. Anderson. Feather Trace: A light-weight event tracing toolkit. In *OSPERT*, pages 19–28, 2007.

[11] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. Strict Fibonacci heaps. In *STOC*, pages 1177–1184. ACM, 2012.

[12] A. Burns, R. I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a c=d task splitting scheme. *Real-Time Systems*, 48(1):3–33, 2012.

[13] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*, pages 111–123, 2006.

[14] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *ECRTS*, pages 247–258, 2007.

[15] F. Cerqueira, A. Gujarati, and B. B. Brandenburg. Linux’s processor affinity API, refined: Shifting real-time tasks towards higher schedulability. In *RTSS*, pages 249–259, 2014.

[16] F. Cerqueira, M. Vanga, and B. Brandenburg. Scaling global scheduling with message passing. In *RTAS*, pages 263–274, 2014.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[18] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011.

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

[20] P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, 2010.

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

[22] 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.

[23] A. Gujarati, F. Cerqueira, and B. B. Brandenburg. Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities. In *ECRTS*, pages 69–79, 2013.

[24] A. Gujarati, F. Cerqueira, and B. B. Brandenburg. Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. *Real-Time Systems*, 51(4):440–483, 2015.

[25] P. Hall. On representatives of subsets. *J. London Math. Soc.*, 10:26–30, 1935.

[26] H.-C. Jang and H.-W. Jin. Miami: Multi-core aware processor affinity for TCP/IP over multiple network interfaces. In *HOTI*, 2009.

[27] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *ECRTS*, pages 249–258, 2009.

[28] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, 5(4): 379 – 400, 1994.

[29] G. Muratore, U. M. Schwarz, and G. J. Woeginger. Parallel machine scheduling with nested job assignment restrictions. *Operations Research Letters*, 38(1):47 – 50, 2010.

[30] B. Nikolic, K. Bletsas, and S. M. Petters. Priority assignment and application mapping for many-cores using a limited migrative model. Technical Report TR-140204, CISTER, 2014.

[31] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

[32] J. D. Salehi, J. F. Kurose, and D. F. Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version). *IEEE/ACM Trans. Netw.*, 4(4):516–530, 1996.

[33] A. Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency*. Springer, 2003.