Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities

Sanjoy Baruah
The University of North Carolina
baruah@cs.unc.edu

Björn Brandenburg
Max Planck Institute for Software Systems
bbb@mpi-sws.org

***Abstract*—In many current multiprocessor real-time operating systems, programmers have the ability to set *affinity masks* that pin a process to a specified subset of the processors in the system. Given a real-time task system consisting of a collection of implicit-deadline sporadic tasks with an affinity mask specified for each task that is to be implemented upon an identical multiprocessor platform, this paper addresses the question of determining whether the task system can be implemented upon the platform to always meet all deadlines, while respecting the affinity mask restrictions. An algorithm is derived that answers this question efficiently in run-time that is polynomial in the representation of the task system.**

I. Introduction

Many current multiprocessor operating systems provide facilities that allow programmers to set *affinity masks* for processes or threads, specifying the processor[s] upon which the process is permitted to execute – examples include `SetThreadAffinityMask` in Windows and the `sched_setaffinity` function in Linux. The programmer may set these affinity masks to achieve a wide variety of goals including improved performance (by, e.g., co-locating processes that communicate a lot or are likely to share cache), load balancing, inter-component isolation, spatial separation of replicated components for fault tolerance, etc. These affinity masks are finding use in soft and hard real-time systems as well, and are now supported by many contemporary multiprocessor real-time operating systems such as VxWorks, LynxOS, QNX, and real-time variants of Linux — see, e.g., [4] for a discussion on the use of this feature in real-time systems. The ready availability of affinity mask functionality in RTOSes, and their increasing use in real-time systems, indicate that real-time scheduling theory should devote some attention to studying the problem of scheduling systems with affinity masks specified.

In this research, we address the issue of *feasibility analysis* of such systems upon identical multiprocessor platforms. We assume that the real-time system under analysis may be modeled as a collection of simple recurring (periodic/ sporadic) preemptive tasks [9], and seek algorithms to determine whether the system can be executed on the processors in such a manner that all timing constraints are met, while respecting the affinity mask constraints. We make the assumptions that processor *preemption* is permitted and incurs no cost, and that *global scheduling* within the restrictions imposed by the affinity masks is allowed (i.e., a preempted job of a task may resume execution on a different processor upon which it is permitted to execute according to its affinity mask). We have two significant contributions to report: **(i)** a polynomial-time algorithm that solves this problem (and in fact, actually constructs a preemptive schedule); and **(ii)** a proof that the schedules so produced in fact end up executing most tasks on just a single processor, with relatively few tasks needing to migrate from one processor to another.

Organization. The remainder of this paper is organized as follows. In Section II, we formally define the problem that we wish to solve, and briefly describe related research. In Section III we show that our problem may be considered to be a special case of a more general problem – global scheduling on unrelated multiprocessors – that is known to be solvable in polynomial time; this immediately implies that our problem, too, is solvable in polynomial time. In Section IV we derive an algorithm for directly solving our problem (rather than by transforming it to a problem of global scheduling on unrelated multiprocessors); we show that this is a simpler algorithm, and further yields a scheduling strategy that can be used as a dispatcher during run-time. We will see that the run-time efficiency of this dispatcher depends upon the number of tasks that execute upon multiple processors; accordingly in Section V we derive a bound on the number of such tasks in the scheduling strategies that are generated by our algorithm. We conclude in Section VI, with a summary of the results presented here and a brief discussion on ongoing research into extensions to these results.

II. Task model and related work

We restrict our attention to systems of *implicit-deadline sporadic tasks* [9]. That is, we assume that the real-time workload can be modeled as a finite collection of sporadic tasks, with each task characterized by a *worst-case execution requirement (WCET)* and a *period*. Such a sporadic task τ_i with WCET C_i and period T_i is assumed to generate a potentially infinite sequence of *jobs* that need to be executed, with each job needing at most C_i units of execution within T_i time-units of the job's generation, and successive jobs being generated at least T_i time-units apart. We assume that different jobs, whether generated by the same or different tasks, are completely *independent* of one another in the sense that they do not share resources, nor do they have data dependencies.

We also assume that our execution model allows for processor *preemption*, i.e., a job executing on a processor may be interrupted at any instant in time, and its execution resumed later, at no cost or penalty.

Let τ denote an implicit-deadline sporadic task system comprised of the n tasks $\tau_1, \tau_2, \ldots, \tau_n$, that is to be implemented upon a multiprocessor platform π consisting of the m identical processors $\pi_1, \pi_2, \ldots, \pi_m$. In addition to its WCET C_i and its period T_i, we assume that task τ_i is further characterized by a *processor affinity* set $\alpha_i \subseteq \{1, 2, \ldots, m\}$, denoting that τ_i may execute on the processors with index in α_i. We further let u_i denote the *utilization* of τ_i, where $u_i = C_i/T_i$.

Restrictions may be placed on the form of the processor affinity sets. For instance, it may be required that, for each task τ_i, $\alpha_i = \{1, 2, \ldots, m\}$ in which case we have the standard *global* schedulability constraint; it may be required that, for each task τ_i, $|\alpha_i| \leq k$ for some constant k (if $k = 1$ then we have standard *partitioned* scheduling with the task partitioning already specified); it may be mandated that, for all i, j, it is the case that either **(i)** $\alpha_i = \alpha_j$, or **(ii)** $\alpha_i \cap \alpha_j = \emptyset$ (i.e., we are specifying *clustered* scheduling); etc. In this paper we assume for the most part that the processor affinity sets are not restricted in any manner: i.e., we consider *arbitrary processor affinity **(APA)*** scheduling.

An APA scheduling *instance* is specified as an ordered pair (τ, π); given such an instance, the *APA feasibility analysis problem* asks whether it is possible to schedule τ upon π in a manner that respects the processor affinity restrictions, such that all jobs of all the tasks in τ meet all their deadlines, for all sequences of jobs that may legally be generated by the tasks in τ.

Related work. While there has been considerable work addressing the use of affinity masks to achieve programmer goals (e.g., [3] explores their use in boosting TCP performance on multiprocessor servers), we are not aware of any prior research on the subject of this paper, which is providing scheduling support for affinity masks that are already specified by the programmer. Previous work in real-time scheduling theory that addresses processor affinity at all (such as [12], [13], [11]) instead seem to be primarily centered on attempts at assigning jobs to processors upon which they had executed previously.

In our opinion, the work discussed in [4], which describes itself as *an initial step in the development of real-time scheduling theory for APA scheduling*, is closest in spirit to the work discussed in this paper – we look upon the current paper as a further step in this development of a theory for APA scheduling. However, [4] does not consider feasibility analysis; instead, an interesting aspect to the use of affinity masks is considered: is it possible to *enhance* schedulability – i.e., render an otherwise unschedulable system schedulable – by appropriately setting affinity masks? Results in [4] show that for job-level fixed priority scheduling policies (such as EDF), there are task systems that are not partitioned or global schedulable, but that can be made schedulable by appropriately setting affinity masks.

III. APA feasibility analysis

All the processors in an identical multiprocessor platform are assumed to have the same computing capacity, in the sense that the amount of execution completed by executing a job on a processor for a given amount of time is exactly the same on all the processors. By contrast in an *unrelated* multiprocessor platform, there is an execution rate $r_{i,j}$ associated with each task-processor pair with the interpretation that a job of the i'th task completes $(r_{i,j} \times \delta)$ units of execution by executing on the j'th processor for a duration δ. It is evident that APA feasibility analysis is a special case of global feasibility analysis on unrelated multiprocessors: given an instance (τ, π) of the APA feasibility analysis problem, we can obtain an instance of the problem of global feasibility analysis on unrelated multiprocessors by having the same set of tasks and processors, and for each (i, j) setting the execution rate $r_{i,j}$ for task τ_i on processor π_j according to the following rule:

$$r_{ij} = \begin{cases} 1, & \text{if } j \in \alpha_i \\ 0 & \text{otherwise} \end{cases}$$

It has been shown [8] that the problem of determining whether a given implicit-deadline sporadic task system is feasible under global scheduling upon an unrelated multiprocessor platform can be solved in polynomial time[1]. Since (as we saw above) APA scheduling is a special case of scheduling on unrelated processors, this immediately leads to the conclusion that

Theorem 1: *Determining whether a given APA scheduling instance (τ, π) is APA-feasible can be determined in time polynomial in the representation of (τ, π).*

IV. A scheduling algorithm

Theorem 1 above showed that APA feasibility analysis is a polynomial-time problem, by transforming APA feasibility analysis to global feasibility analysis on unrelated multiprocessors. One could therefore obtain a polynomial-time algorithm for APA feasibility analysis by simply applying this transformation. In this section, we describe a more efficient algorithm that does APA feasibility analysis directly without first transforming the problem to one on unrelated multiprocessors; furthermore, our algorithm enables us to obtain a scheduling strategy that performs the actual scheduling during run-time. Although this algorithm is based on the approach presented in [8], it is simpler than the algorithm in [8]: Section IV-D explains how our algorithm for APA-scheduling upon identical processors is simpler that the algorithm in [8] for global scheduling upon unrelated processors.

In the remainder of this section let (τ, π) denote an APA instance in which τ consists of the n tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$, and π consists of the m identical unit-speed processors $\pi_1, \pi_2, \ldots, \pi_m$.

[1]Actually, [8] was concerned with the scheduling of collections of independent jobs rather than systems of recurrent tasks, but the extension to implicit-deadline sporadic task systems is straightforward – see, e.g., [1].

Given this APA instance (τ, π), our scheduling strategy proceeds in three steps:

1) First, we construct a linear program that has a solution (is *feasible*, in the language of linear programming) if and only if (τ, π) is APA-feasible – how we do this is described in Section IV-A.
2) We then solve this linear program, and use the solution to construct a schedule "template" — this step is described in Section IV-B.
3) In Section IV-C, we describe how we then use this schedule template to actually construct a schedule during run-time.

We will use the following example throughout this section, to illustrate our algorithm.

Example 1: Our APA instance (τ, π) has $\tau = \{\tau_1, \tau_2, \tau_3\}$ with parameters as given below, while $\pi = \{\pi_1, \pi_2\}$.

τ_i	$\mathbf{C}_i$	$\mathbf{T}_i$	α_i	$\mathbf{u}_i$
τ_1	7	10	$\{1\}$	0.7
τ_2	6	10	$\{2\}$	0.6
τ_3	10	20	$\{1, 2\}$	0.5

A. Constructing a linear program

As a first step in obtaining a run-time scheduling algorithm, we will represent APA feasibility as a linear programming problem. We define $n \times m$ real-valued variables $x_{i,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, with variable $x_{i,j}$ denoting the fraction of τ_i that executes on processor π_j. We constrain each $x_{i,j}$ variable to be non-negative: $x_{i,j} \geq 0$ for all i, j. (While we could also constrain each $x_{i,j}$ to be no larger than one, specifying such a constraint turns out to be unnecessary in the sense that it is implied by other constraints that we will add to the linear program.)

We now seek to enforce the requirement that *each task receives its appropriate amount of execution time*. Since the affinity mask of task τ_i restricts the processors upon which τ_i may execute, this requirement can be represented by n equations of the following form, one for each $i = 1, 2, \ldots, n$:

$$\sum_{j \in \alpha_i} x_{i,j} = 1 \tag{1.1}$$

Next, we seek to enforce the requirement that *the capacity constraint of each processor is met*. This is represented by m inequalities of the following form, one for each $j = 1, 2, \ldots, m$:

$$\sum_{i=1}^{n} (u_i x_{i,j}) \leq 1 \tag{1.2}$$

The resulting LP problem incorporating these two sets of constraints for any APA instance (τ, π), over the $(n \times m)$ variables $\{x_{i,j}\}_{i=1,\ldots,n; j=1,\ldots,m}$ which are all constrained to being ≥ 0 and ≤ 1, is depicted in Figure 1.

Example 2: The linear program for the instance of Example 1 is

$$\begin{aligned}
x_{11} &= 1 \\
x_{22} &= 1 \\
x_{31} + x_{32} &= 1 \\
0.7\, x_{11} + 0.5\, x_{31} &\leq 1 \\
0.6\, x_{22} + 0.5\, x_{32} &\leq 1
\end{aligned}$$

where the first three constraints correspond to instantiations of Constraint (1.1) for tasks τ_1, τ_2, and τ_3 respectively, while the last two correspond to instantiations of Constraint (1.2) for processors π_1 and π_2.

(Note that this linear program has no objective function specified: we are simply seeking to determine *any* assignment of values to the x_{ij} variables that would cause all the constraints to evaluate to true.)

It may be verified that the following assignment of values to the x_{ij} variables

$$x_{11} = 1; x_{22} = 1; x_{31} = 0.4; \text{ and } x_{32} = 0.6 \tag{2}$$

constitutes a possible solution to this linear program. ■

It is immediately evident that the linear program APA-Feas(τ, π) has a solution if the instance (τ, π) is APA-feasible; this is formally demonstrated in Lemma 1 below.

Lemma 1: *If instance (τ, π) is APA-feasible, then APA-Feas(τ, π) has a solution.*

Proof: Since (τ, π) is APA-feasible, it is possible to generate a schedule for any sequence of jobs that could legally be generated by this instance. Let us therefore consider the sequence of job arrivals in which jobs of each task τ_i arrive at all instants $k \cdot T_i$, for all $k = 0, 1, 2, \ldots$, and executes for exactly C_i time units. Let P denote the least common multiple (lcm) of the periods of the n tasks.

Consider a schedule for this sequence of job arrivals, in which all job deadlines are met. Let $f_{i,j}$ denote the fraction of the total amount of time over $[0, P)$ during which task τ_i executed on processor π_j, for each i and each j. We will show that all the constraints in APA-Feas(τ, π) are satisfied when $x_{i,j} \leftarrow (f_{i,j}/u_i)$ for each i, j.

Constraints (1.1): The total amount of execution received by task i over $[0, P)$ is given by

$$\sum_{j \in \alpha_i} (f_{i,j} \cdot P) \;=\; P \sum_{j \in \alpha_i} (f_{i,j}) \;.$$

There are $\frac{P}{T_i}$ jobs of task τ_i with arrival times and deadlines in $[0, P)$; in a schedule in which all job deadlines are met, τ_i will therefore have received $C_i \cdot \frac{P}{T_i}$ units of execution over

APA-Feas(τ, π)
Determine non-negative values for the variables $\{x_{i,j}\}, 1 \le i \le n, 1 \le j \le m$, satisfying the following constraints:

$$\begin{aligned} &\sum_{j \in \alpha_i} x_{i,j} = 1 \;, \qquad (i = 1, 2, \ldots, n) && (1.1) \\ &\sum_{i=1}^{n} \big(u_i \, x_{i,j}\big) \le 1 \;, \qquad (j = 1, 2, \ldots, m) && (1.2) \end{aligned} \tag{1}$$

Fig. 1. Linear Programming formulation of APA feasibility.

$[0, P)$. That is,

$$\begin{aligned} P \sum_{j \in \alpha_i} (f_{i,j}) &= C_i \cdot \frac{P}{T_i} \\ \Leftrightarrow \quad \sum_{j \in \alpha_i} (f_{i,j}) &= \frac{C_i}{T_i} \\ \Leftrightarrow \quad \sum_{j \in \alpha_i} \left(\frac{f_{i,j}}{u_i}\right) &= 1 \end{aligned}$$

Thus, Constraint (1.1) of APA-Feas(τ, π) is satisfied when $x_{i,j} \leftarrow (f_{i,j}/u_i)$ for all i, j.

Constraints (1.2): For each j, this simply reflects the fact that processor π_j could not have executed for more than P time units over the interval $[0, P)$.

That completes the proof of Lemma 1. ■

Lemma 1 above showed that if the instance (τ, π) is APA-feasible then the linear program APA-Feas(τ, π) of Figure 1 has a solution. What about in the opposite direction? — does a solution to the LP imply that (τ, π) is APA-feasible? It turns out that the answer is "yes;" in Sections IV-B and IV-C below we will show how to use a solution to the linear program APA-Feas(τ, π) to develop a strategy for the run-time scheduling of APA instance (τ, π). Taken in conjunction with Lemma 1, this will serve to show that the linear program APA-Feas(τ, π) of Figure 1 has a solution if and only if (τ, π) is APA-feasible.

B. Constructing a schedule template

Once we have constructed the linear program APA-Feas(τ, π) as discussed above, we determine whether it has a solution or not. If not, Lemma 1 tells us that instance (τ, π) is not APA-feasible; we therefore declare failure and stop. If APA-Feas(τ, π) is feasible, however, we show below how we can use any feasible solution to APA-Feas(τ, π) to determine a scheduling strategy for the APA-instance (τ, π).

Let us therefore suppose that APA-Feas(τ, π) is a feasible linear program, and let $\{x_{i,j}\}$ denote a solution to it. We now describe how we can use a feasible solution to APA-Feas(τ, π) to construct a schedule template for the tasks in τ upon the processors in π, such that all deadlines are met.

Let ℓ be defined as follows:

$$\ell := \max\Big\{\max_{i=1}^{n}\{\sum_{j \in \alpha_i} u_i x_{i,j}\}, \max_{j=1}^{m}\{\sum_{i=1}^{n} u_i x_{i,j}\}\Big\} \tag{3}$$

Example 3: From the LP solution in Expression 2, we compute ℓ for the APA-instance of Example 1 according to Expression 3 as follows:

$$\begin{aligned} \ell &= \max\Big\{\max\{0.7, 0.6, 0.2 + 0.3\}, \\ &\qquad \max\{0.7 + 0.2, 0.6 + 0.3\}\Big\} \\ &= \mathbf{0.9} \end{aligned} \tag{4}$$

■

Notice that by Constraint (1.1) of APA-Feas(τ, π), $\left(\sum_{j \in \alpha_i} u_i x_{i,j}\right)$ is equal to C_i/T_i. Thus ℓ is set equal to the larger of (i) the maximum utilization of any task, and (ii) the maximum fraction of the computing capacity of any processor, that is committed to executing the tasks in τ. It is evident that $\ell \le 1$ for any feasible solution to APA-Feas(τ, π).

We will now apply the technique introduced in [8] to construct a schedule over the interval $[0, \ell)$ that, for all i, j, executes task τ_i for exactly $(u_i x_{i,j})$ time units on processor π_j. By Constraint (1.1) of APA-Feas(τ, π), this implies that each τ_i receives exactly (C_i/T_i) units of execution over $[0, \ell)$. (Since $\ell \le 1$, it follows that this schedule completes at or before time-instant 1; hence, a schedule in which all job deadlines are met may be obtained by repeating this schedule over all time-intervals $[k, k+\ell)$, for all integer k. However, this is *not* how we will be constructing our schedule in run-time – the manner in which we do so is described in Section IV-C.)

In constructing the schedule over $[0, \ell)$, the major challenge is to ensure that no *job-level parallelism* occurs: at each instant, each task τ_i should execute on at most one processor π_j. Our approach towards constructing such a schedule is iterative: we repeatedly choose a subset of the set of tasks for execution, the processors to execute them on, and the length of time for which to execute them. More specifically, in each iteration we

1) Determine a one-to-one mapping χ from a subset of the set of tasks to a subset of the set of processors — if

task τ_i is in this subset chosen for execution, then τ_i is executed on $\pi_{\chi(i)}$ over the interval $[\ell - \delta, \ell)$ for some δ, $0 < \delta \leq \ell$.
(The precise manner in which the mapping χ, and the value of δ, are determined, is described below.)

2) For each (i, j) pair such that $\chi(i) = j$, we decrement $x_{i,j}$ by an amount (δ/u_i):
$$x_{i,j} \leftarrow x_{i,j} - \frac{\delta}{u_i}$$
3) We decrement ℓ by an amount δ: $\ell \leftarrow \ell - \delta$.

During each such iteration, we will choose the tasks to be executed over $[\ell - \delta, \ell)$ in such a manner that the values of $x_{i,j}$ and ℓ after the iteration will continue to satisfy Equation 3. Since $\delta > 0$, it follows that executing these three steps repeatedly will yield the desired schedule.

This process is illustrated in Example 4 below for the APA instance of Example 1. The reader is encouraged to glance through Example 4 at the current moment and observe that the algorithm undergoes three iterations for that example. The first iteration maps tasks τ_1 and τ_3 on processors π_1 and π_2 respectively for 0.3 time units; the second iteration maps tasks τ_3 and τ_2 on processors π_1 and π_2 respectively for 0.2 time units; and the third (and final) iteration maps tasks τ_1 and τ_2 on processors π_1 and π_2 respectively for 0.4 time units.

Determining χ and δ

As stated above, our approach towards constructing the schedule template is iterative in the sense that we choose a subset of the set of tasks for execution, the processors to execute them on, and the length of time for which to execute them, during each iteration. We now describe each iteration of this iterative process, starting with a high-level overview and following up with a more detailed description.

Overview. With respect to the *current iteration*[2], we define a processor π_j to be ***full*** if $\sum_{i=1}^{n} u_i x_{i,j} = \ell$, thereby requiring that processor π_j must be continually busy over the interval $[0, \ell)$ in a correct schedule. We define a task τ_i to be ***urgent*** if $\sum_{j \in \alpha_i} u_i x_{i,j} = \ell$; this requires that task τ_i must execute continually over the interval $[0, \ell)$ in a correct schedule. (We observe from the definition of ℓ in Equation 3 that there must be some full processor[s] and/ or some urgent task[s] at the start of the first iteration). In each iteration, we obtain a one-to-one mapping χ from a subset of the set of tasks to a subset of the set of processors, such that each urgent task is mapped upon some processor, and all full processors have some task mapped upon them; in addition, some non-urgent tasks may be mapped upon processors, and some non-full processors may have tasks mapped upon them. Each task τ_i that is so mapped will be executed upon the processor $\pi_{\chi(i)}$ to which it is mapped, for an interval of time δ. That is, task τ_i is executed on processor $\pi_{\chi(i)}$ over the interval $[\ell - \delta, \ell)$, for all task-processor pairs $(i, \chi(i))$ in the mapping.

At the end of the iteration, we decrement the $x_{i,j}$ value corresponding to each such task-processor pair by an amount (δ/u_i), and set $\ell \leftarrow \ell - \delta$. Then, we proceed to the next iteration which involves identifying a new mapping such that all tasks which are urgent (according to these updated values for ℓ and $x_{i,j}$'s), and all processors which are full (also according to these updated values), are mapped. Notice that the set of urgent tasks/ full processors in this next iteration will include all urgent tasks/ full processors of the current iteration, and may contain some additional tasks/ processors.

Details. In each iteration, we will construct a bipartite graph based upon the current values of the $x_{i,j}$'s and ℓ. We will then use the following well-known result concerning matchings in bipartite graphs to determine a matching on the bipartite graph we construct, and use this matching to obtain the mapping χ of a subset of the tasks to a subset of the processors.

For any set of vertices V, let $N(V)$ denote the neighborhood of V – i.e., all vertices that are connected by an edge to some vertex in V. We will make use of Halls' famous condition for *saturating matchings* [5] (see, e.g., [15, page 110] for a text-book exposition).

Theorem 2 (Hall's theorem [5]): *Let $G = (X \bigcup Y, E \subseteq X \times Y)$ denote a bipartite graph with vertex set $(X \bigcup Y)$, and edge-set E. There is a matching in G which saturates X (i.e., in which each vertex in X is matched), if and only if $(|N(S)| \geq |S|)$ for all $S \subseteq X$.*

Our bipartite graph is constructed as follows. The graph has $(n + m)$ vertices, one corresponding to each task τ_i and one to each processor π_j, and an edge (τ_i, π_j) if and only if the value of $x_{i,j}$ during the current iteration is > 0.

Fact 1: *Every collection of vertices corresponding to full processors and every collection of vertices corresponding to urgent tasks in this bipartite graph has "many" neighbors. More precisely,*

1) For any set Γ' of (vertices corresponding to) urgent tasks, it is the case that $|N(\Gamma')| \geq |\Gamma'|$.
2) For any set Π' of (vertices corresponding to) full processors, it is the case that $|N(\Pi')| \geq |\Pi'|$.

Proof: We prove the first statement above; the proof of the second is virtually identical.

Let Γ' denote the vertices corresponding to any set of urgent tasks. Since all tasks in Γ' are urgent, it follows that $\sum_{\tau_i \in \Gamma'} \sum_{\text{all } j} x_{i,j} = |\Gamma'| \cdot \ell$.

By definition, all non-zero $x_{i,j}$'s from all task-vertices $\tau_i \in \Gamma'$ lead to vertices $\pi_j \in N(\Gamma')$. Therefore $\sum_{\tau_i \in \Gamma'} \sum_{\text{all } j} x_{i,j} = \sum_{\tau_i \in \Gamma'} \sum_{\pi_j \in N(\Gamma')} x_{i,j}$. Now if $|N(\Gamma')| < |\Gamma'|$, then it must be the case that, for some $\pi_j \in N(\Gamma')$, $\sum_i x_{i,j} > \ell$. However, having $\sum_i x_{i,j}$ exceed ℓ violates Equation 3 – a contradiction.

■ **End proof** (of **Fact 1**).

[2]At the beginning, the first iteration is the current iteration; consequently, the values of the $x_{i,j}$'s and of ℓ referred to in this paragraph during this first iteration are the values obtained by solving the LP problem APA-Feas(τ, π) and Equation 3 respectively. The values of ℓ and the $x_{i,j}$'s in subsequent iterations are obtained as described in the next paragraph (the one starting with "At the end of the iteration…").

Let Γ denote the set of all vertices corresponding to tasks, and Π the set of all vertices corresponding to processors, in this bipartite graph. Let $\Gamma_u \subseteq \Gamma$ denote the set of all vertices corresponding to urgent tasks, and $\Pi_f \subseteq \Pi$ the set of all vertices corresponding to full processors.

1) Determine a matching from all vertices in Γ_u to a subset of the vertices in Π – by Fact 1 and Theorem 2, this can always be done.
2) Determine a matching from all vertices in Π_f to a subset of the vertices in Γ – by Fact 1 and Theorem 2, this can always be done.
3) If an urgent task appears in the second matching as well, then discard the edge that this task was matched with in the first matching.
   With the remaining edges, it is evident that each urgent task is matched with exactly one processor, and each full processor is matched with one or two tasks; furthermore, if a full processor is matched with two tasks then exactly one of these is an urgent task.
4) If a full processor remains matched with two tasks with these remaining edges, then discard the edge matching it with a non-urgent task, and retain only that edge that matches it with an urgent task.

It is evident that the remaining edges constitute a partial matching on the bipartite graph that satisfies the following property:

Fact 2: *The union of all the remaining edges is a matching in which* each full processor is matched, and each urgent task is matched. *(There may be additional matched vertices, corresponding to non-full processors and non-urgent tasks, as well.)* ■

The matching obtained above defines the mapping χ from tasks to processors – if (τ_i, π_j) is an edge in the matching, then $\chi(i) \leftarrow j$. By Fact 2, it is ensured that each urgent task is executed, and that each full processor has some task executed upon it. It remains now to specify how long each task τ_i that has been so matched should execute upon processor $\pi_{\chi(i)}$; i.e., how the value of δ is determined. We want the largest value of δ satisfying the following three conditions:

1) For each task-processor pair (τ_i, π_j) such that $\chi(i) = j$, we need $\delta \le u_i x_{i,j}$; this ensures that task τ_i can indeed execute on processor π_j for the δ time units in $[\ell - \delta, \ell)$.
2) For each task τ_i that did not get mapped, we need $\delta \le (\ell - \sum_{j \in \alpha_i} u_i x_{i,j})$; this ensures that task τ_i remains non-urgent throughout the interval $[\ell - \delta, \ell)$.
3) For each processor π_j that did not get mapped, we need $\delta \le (\ell - \sum_{\text{all } i} u_i x_{i,j})$; this ensures that processor π_j remains non-full throughout the interval $[\ell - \delta, \ell)$.

Hence, δ is computed as follows:

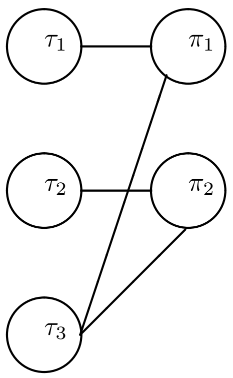
$$\delta = \min \begin{cases} u_i x_{i,j}, & \text{all } (i,j) \text{ such that } \chi(i) = j \\ \ell - \sum_{j \in \alpha_i} u_i x_{i,j}, & \text{all } i \text{ such that } \tau_i \text{ was not selected for execution on any processor} \\ \ell - \sum_{i=1}^{n} u_i x_{i,j}, & \text{all } j \text{ such that } \pi_j \text{ was not selected to execute any task} \end{cases} \tag{5}$$

Thus the current iteration defines a schedule for the interval $[\ell - \delta, \ell)$: each task τ_i which is mapped on to some processor $\pi_{\chi(i)}$ is executed on processor $\pi_{\chi(i)}$ over this interval. The value of ℓ is decremented by an amount δ, and all the $x_{i,j}$'s corresponding to tasks that are scheduled during this interval are each decremented by an amount (δ/u_i), prior to the start of the next iteration.

Example 4: The following solution to the linear program APA-Feas(τ, π) for the APA-instance (τ, π) of Example 1 was given in Expression 2:

$$x_{11} = 1; x_{22} = 1; x_{31} = 0.4; \text{ and } x_{32} = 0.6$$

At the start of the ***first iteration***, the bipartite graph that is constructed contains the edges corresponding to each non-zero $x_{i,j}$, i.e., the edges (τ_1, π_1), (τ_2, π_2), (τ_3, π_1), and (τ_3, π_2).



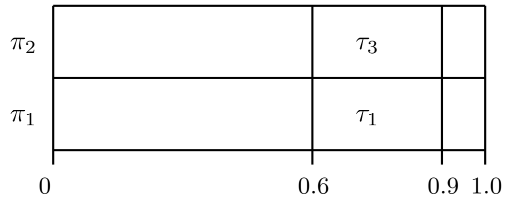
In Example 3, we had determined that the value of ℓ for the APA-instance (τ, π) of Example 1 is 0.9. Since this is larger than $u_1 x_{11}$, $u_2 x_{22}$, and $u_3 x_{31} + u_3 x_{32}$, the set of urgent tasks Γ_u is empty. However, both processors are full (since $u_1 x_{11} + u_3 x_{31} = 0.9$ and $u_2 x_{22} + u_3 x_{32} = 0.9$). Hence a possible matching that would match all full processors would map task τ_1 to processor π_1 and task τ_3 to processor π_2: $\chi(1) = 1, \chi(3) = 2$.

Based on this mapping, we would compute δ according to Equation 5:

$$\delta = \min \begin{cases} u_1 x_{11}, u_3 x_{32} \\ (\ell - u_2 x_{22}) \\ - \end{cases}$$

which evaluates to 0.3 – the value of both $u_3 x_{32}$ and $(\ell - u_2 x_{22})$.

Hence the current iteration defines a schedule over the interval $[\ell - \delta, \ell)$, or $[0.6, 0.9)$:



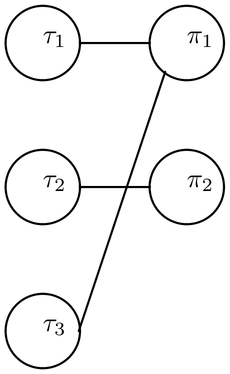
For the next (i.e., ***second) iteration***, the values of the x_{ij}'s are updated to represent the remaining execution requirements:

$$x_{11} = (1 - \frac{0.3}{0.7}) = 4/7; x_{22} = 1; x_{31} = 0.4, \text{ and } x_{32} = 0 \quad (6)$$

while the value of ℓ is also decremented by an amount δ:

$$\ell \leftarrow 0.9 - 0.3 (= 0.6) \quad (7)$$

The bipartite graph that is constructed during this second iteration is



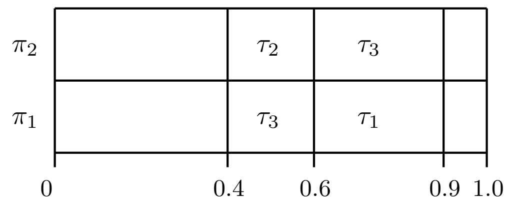
(the edge (τ_3, π_2) is missing since $x_{32} = 0$.)
Since ℓ is equal to the remaining utilization for task τ_2, the set of urgent tasks Γ_u consists of τ_2; both processors are full. Hence a possible matching that would match both processors would map task τ_2 to processor π_2 and task τ_3 to processor π_1: $\chi(2) = 2, \chi(3) = 1$.

Based on this mapping, we would compute δ according to Equation 5:

$$\delta = \min \begin{cases} u_2 x_{22}, u_3 x_{31} \\ (\ell - u_1 x_{11}) \\ - \end{cases}$$

which evaluates to 0.2 (since this is the value of both $u_3 x_{31}$ and $(\ell - u_1 x_{11})$).

Hence this second iteration defines a schedule over the interval $[\ell - \delta, \ell)$, or $[0.4, 0.6)$; adding this to the schedule generated during the first iteration, we get the following:



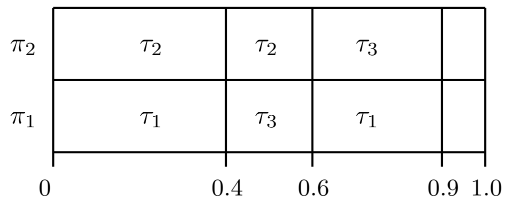
For the next (i.e., ***third) iteration***, the values of the x_{ij}'s are updated to represented the remaining execution requirements:

$$x_{11} = 4/7; x_{22} = (1 - \frac{0.2}{0.6}) = 2/3; x_{31} = 0.0, \text{ and } x_{32} = 0 \quad (8)$$

while the value of ℓ is also decremented by an amount δ:

$$\ell \leftarrow 0.6 - 0.2 (= 0.4) \quad (9)$$

The bipartite graph would have only the two edges (τ_1, π_1) and (τ_2, π_2): both τ_1 and τ_2 are urgent and both π_1 and π_2 are full. Hence χ maps τ_1 to π_1 and τ_2 to π_2, to yield the following schedule:



Although the algorithm described above would stop with this schedule, ***heuristics*** can be applied to this schedule to reduce the number of preemptions. These heuristics observe that task τ_3 is the only one that has a presence on both processors; hence, τ_3's allocation on π_1 can be moved to the beginning of the interval (i.e., to $[0.0, 0.2)$), and on π_2 to the end of the interval (i.e., to $[0.6, 0.9)$); this has the effect of allowing the two allocations of τ_1 on π_1 and of τ_2 on π_2 to be contiguous in time, thereby avoiding preemptions between them. The final schedule template obtained after applying these heuristic optimizations is depicted in Figure 2 below:

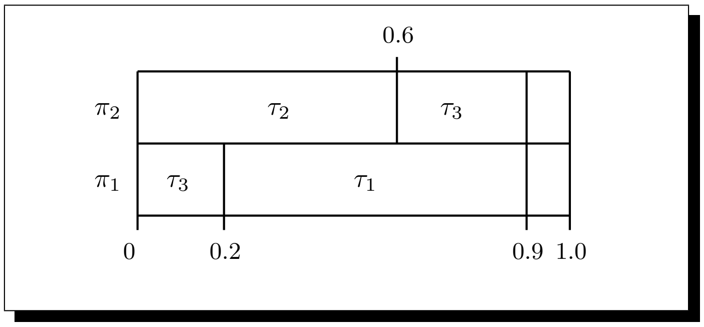


Fig. 2. The schedule template for the instance of Example 1

C. *Runtime scheduling*

Once a schedule template has been constructed as described above, we can use this template to make scheduling decisions at run-time. If we were not concerned with the number of preemptions, this would be trivial: we could divide the time-line into arbitrarily small intervals and simply replicate this template, appropriately scaled in size, during each interval — if some task does not have any job awaiting execution during some time-instant, its allocation could simply idle the processor.

However, it should be evident that such a schedule will experience a very large number of preemptions, and is likely to result in unacceptably high run-time overhead for many (perhaps most) realtime applications. We are therefore working on developing a run-time scheduling policy that only invokes the template upon the release of jobs of tasks that have non-zero execution on more than one processor (i.e., for jobs of tasks τ_i for which there are j_1 and j_2 in α_i, $j_1 \neq j_2$,

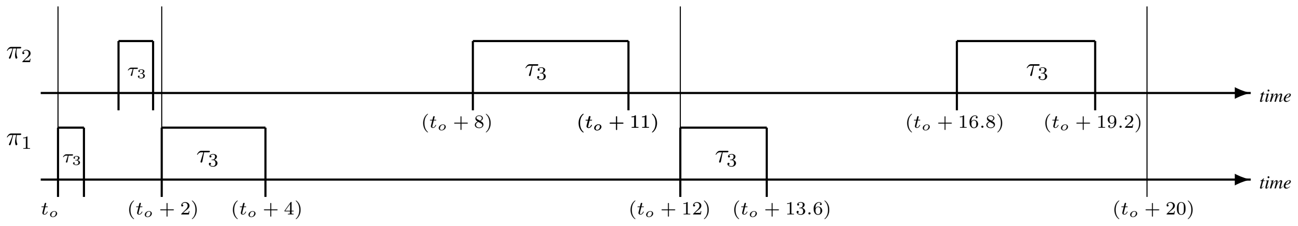


Fig. 3. Example 5: A job of τ_3 arrives at time-instant t_o, with a deadline at time-instant $t_o + 20$. Capacity on processors π_1 and π_2 is reserved for this job as shown. (The time axis is not labeled over $[t_o, t_o + 2)$ to avoid over-cluttering the figure — over this interval, τ_3 has a reservation over $[t_o, t_o + 0.4)$ on π_1 and over $[t_o + 1.2, t_o + 1.8)$ on π_2.)

such that x_{i,j_1} and x_{i,j_2} are both non-zero in the solution to the linear program APA-Feas(τ, π)). When a job of such a τ_i is released, the template is "scaled" by an appropriate factor and reservations are made on all the processors for only those tasks whose jobs may execute upon more than one processor. We stress that it is *not* necessary to make these reservations for tasks that execute upon only one processor: jobs of such tasks may be scheduled upon their respective processors by executing preemptive uniprocessor EDF upon the non-reserved capacities on these processors. Below, we provide a brief description of the operation of this run-time algorithm when there is only one task that executes upon more than one processor; in Example 5 we will illustrate its operation on our example APA instance.

Suppose that a job of task τ_i, that executes upon more than one processor in the schedule template, arrives at some time-instant t_o. This job has a deadline at time-instant t_o+T_i; hence, we will need to make reservations over the interval $[t_o, t_o+T_i)$. Let $d_1, d_2, \ldots, d_k$ denote the (absolute) deadlines, indexed in increasing order (i.e., $d_j < d_{j+1}$ for all j) of jobs that had arrived prior to t_o, are still active at time t_o, and have deadlines within $[t_o, t_o + T_i)$. Let Δ denote (an upper bound on) the smallest relative deadline of any job that may arrive over this interval, and that may "interact" with the scheduling of τ_i's job, by, e.g., executing upon one of the processors on which τ_i executes. (A safe value for Δ is the minimum period of any task in the instance, although larger values may be obtained by more careful analysis — we postpone consideration of an optimal choice for Δ to future work.) To determine the reservations that must be made, we will

- First, let $d_o := t_o$. For each value of j, $0 \le j < k$, scale the schedule template by a factor of $(d_{j+1} - d_j)$, and invoke the reservations of this scaled schedule template over the interval $[d_j, d_{j+1})$.
- Next, scale the schedule template by a factor Δ, and invoke the reservations of this scaled schedule template $\lfloor (T_i - d_k)/\Delta \rfloor$ times contiguously beginning at time-instant d_k.
- Finally, scale the schedule template by a factor $((T_i - d_k) \bmod \Delta)$, and invoke the reservations of this scaled schedule template once, over the interval $[t_o+T_i-((T_i-d_k) \bmod \Delta), t_o + T_i)$.

We now illustrate this process on our running example.

Example 5: Recall that we have constructed the template depicted in Figure 2 for the example APA instance of Example 1. Suppose that during run-time we have made the correct scheduling decisions until some time-instant t_o. Suppose that a job of τ_3 arrives at time t_o (and hence has a deadline at time $t_o + 20$). Suppose that there is one other job active at time-instant t_o with deadline at time-instant t_o+2; i.e., $k = 1$ and $d_1 = t_o + 2$. Since the smallest period of any task in the instance is $\min\{10, 10, 20\} = 10$, the parameter Δ may safely be assigned the value 10.

- The schedule template of Figure 2 is first scaled by a factor of $(d_1 - d_o) = (d_1 - t_o)$, or 2; this template yields τ_3's reservations over $[t_o, t_o + 2)$.
- Next, the schedule template is scaled by a factor of Δ, or 10; this template is repeated $\lfloor (T_i - d_k)/\Delta \rfloor = \lfloor (20 - 2)/10 \rfloor$ times, i.e., once, to obtain τ_3's reservations over the interval $[t_o + 2, t_o + 12)$.
- Finally, it is scaled by a factor $((T_i - d_k) \bmod \Delta) = ((20 - 2) \bmod 10) = 8$, to obtain the reservations over the interval $[t_o + 12, t_o + 20)$.

The resulting reservations are illustrated in Figure 3. ■

Since the schedule template must be used only upon the arrival of jobs of tasks that execute on more than one processor, and capacity reserved for only such tasks, it is to be expected that the efficiency of this run-time scheduler is highly dependent upon how many tasks execute on more than one processor — in the extreme case where each task executes on only a single processor, we do not need to use the template at all and may simply execute the uniprocessor EDF dispatcher upon each processor. Later in Section V, we will bound the number of tasks that execute upon more than one processor in the schedule templates that are generated by the algorithm described in Section IV-B.

D. Comparison with global feasibility analysis upon unrelated multiprocessors [8]

At the start of this section, we had stated that although APA feasibility analysis can be considered to be a special case of the problem of global feasibility analysis upon unrelated multiprocessors, it is in fact a simpler problem; our solution as described in Sections IV-A to IV-C above is therefore simpler than the algorithm in [8] for global feasibility analysis on unrelated multiprocessors. We now briefly enumerate a couple of salient differences between our algorithm and the one in [8].

1) Although the approach of [8] also requires the construction of a linear program with $(n \times m)$ x_{ij} variables, the interpretation of these variables is different. Specifically, x_{ij} denotes the *fraction of time* for which task τ_i is executing on processor π_j in the approach of [8], while for us x_{ij} denotes the fraction of task τ_i that is assigned to processor π_j. (Hence in [8] x_{ij} is constrained to lie in the range $[0, u_i]$, rather than $[0, 1]$ as is the case here.)
2) The linear program constructed in [8] had n more constraints than APA-Feas(τ, π): in addition to constraints that are analogous to Constraints (1.1) and (1.2), there were n additional constraints that needed to be explicitly added in order to enforce the prohibition that a single job may not be executing concurrently upon multiple processors.

V. Characterizing the degree of migration

As we saw in Section IV-C above, run-time scheduling is more difficult for tasks that may execute on multiple processors: jobs of such tasks need per-processor "reservations" to be maintained during runtime, whereas tasks that execute on just a single processor may have their jobs scheduled using uniprocessor EDF. It is well known (see, e.g., [10]) that preemptive uniprocessor EDF can be implemented very efficiently to have a run-time that is logarithmic in the number of tasks per scheduling decision, while maintaining reservations for individual jobs requires significant book-keeping and is typically implemented using algorithms that have run-time linear in the number of reservations that are made.

We will now show that the preemptive schedule obtained by the technique described in Section IV *executes most tasks on a single processor only, and bounds the total number of tasks that execute on more than one processor*. More precisely, let us say that a task τ_i has a **presence** on a processor π_j in the schedule obtained using the approach described above, if and only if task τ_i executes for any amount on processor π_j in this schedule (i.e., if $x_{i,j} > 0$ in the solution to the linear program APA-Feas(τ, π) of Figure 1). Intuitively speaking, the total number of presences is a measure of the "degree" to which the schedule is global rather than partitioned; the number of presences in a purely partitioned schedule is equal to the number of tasks n, while a schedule in which each task has a presence on each processor upon which it is allowed to execute would have $\left(\sum_{i=1}^{n} |\alpha_i|\right)$ presences. We will prove in this section that

1) The total number of presences exceeds the total number of tasks by at most m (Lemma 2 below); and
2) At least $(n - m)$ tasks have a presence on a single processor only (Lemma 3 below).

We first state without proof a basic fact concerning linear programming: a proof may be found in any standard text on linear programming (see, e.g., [14]).

Fact 3: *Consider a linear program on N variables, in which each variable is constrained to be non-negative. Suppose that there are M additional linear constraints. If $M < N$, then at most M of the variables have non-zero values at each vertex of the feasible region*[3] *of the linear program.*

Observe that APA-Feas(τ, π) is a linear program on $(n \times m)$ variables that are each constrained to be non-negative, and $(n + m)$ constraints. By Fact 3, therefore, at most $(n+m)$ of the $x_{i,j}$ variables have non-zero values at each vertex of the feasible region of APA-Feas(τ, π). Lemma 2 follows immediately:

Lemma 2: *The total number of presences exceeds the total number of tasks by at most m.*

A crucial observation is that each of the n Constraints (1.1) of APA-Feas(τ, π) is on a *different* set of $x_{i,j}$ variables – the first such constraint has only the variables $x_{1,1}, x_{1,2}, \ldots, x_{1,m}$, the second has only the variables $x_{2,1}, x_{2,2}, \ldots, x_{2,m}$, the i'th has only the variables $x_{i,1}, x_{i,2}, \ldots, x_{i,m}$, and so on. Since there are at most $(n + m)$ non-zero $x_{i,j}$ values, it follows from the pigeon-hole principle that at most m of these constraints may have more than one non-zero variable. For the remaining constraints, the sole non-zero $x_{i,j}$ variable must take on the value 1 in order that the corresponding constraint be satisfied, which implies that the entire task τ_i is assigned to the processor π_j. Lemma 3 follows.

Lemma 3: *For at least $(n - m)$ of the integers i in $\{1, 2, \ldots, n\}$, $\underline{\text{exactly}}$ one of the variables $\{x_{i,1},\ x_{i,2},\ \ldots,\ x_{i,m}\}$ is non-zero in any vertex solution to APA-Feas(τ, π). That is, at least $(n - m)$ tasks will have a presence on one processor only.*

From a computational complexity perspective, determining a vertex in the feasible region of a linear program is an easy problem. Many LP solvers return vertex solutions; others that are based on interior-point algorithms [6] or ellipsoid algorithms [7] do not guarantee to find a vertex point (although the simplex algorithm [2] does). There are efficient polynomial-time algorithms (see, e.g, [14]) for obtaining a vertex optimal solution given any non-vertex optimal solution to a LP problem – if the LP-solver being used does not guarantee to return a vertex-optimal solution, then one of these algorithms may be used to obtain a vertex-optimal solution from the feasible solution that is returned by the LP-solver.

VI. Context and conclusions

The ability to specify processor affinity masks for individual tasks is now widely available in real-time operating systems; as a consequence, there has been an increasing use of affinity masks by programmers of real-time systems to achieve diverse goals including improved performance, load-balancing, inter-component isolation, physical separation for fault-tolerance, etc. As the ability to set processor affinity masks comes to be

[3]The *feasible region* in N-dimensional space for this linear program is the region over which all $(N + M)$ constraints – the M linear constraints plus the N constraints that each of the N variables be non-negative – are satisfied.

increasingly used in real-time systems, we believe it important that the real-time scheduling theory community seek to obtain a better understanding of this phenomenon.

When a new model is investigated by the real-time scheduling theory community, feasibility-analysis and run-time scheduling have traditionally been the first scheduling problems studied. In this paper, we have studied the problems of determining whether a real-time system with specified affinity masks is feasible or not and if feasible, of obtaining a run-time scheduling strategy that meets all timing constraints. We have shown that the feasibility analysis problem can be solved in polynomial time, by transforming it to the earlier-studied problem of preemptive global scheduling on unrelated multiprocessors. We have also derived an algorithm that does feasibility-analysis directly from first principles; this algorithm leads to a run-time scheduling strategy. We demonstrated the efficiency of this run-time strategy by showing that it can be implemented efficiently if most tasks execute on just a single processor, and then proving that the schedules generated by our scheduling strategy indeed possess this property.

Our work can be extended in several directions. One obvious extension would be to consider more general workload models: to begin with, it would be useful to have results concerning task systems in which relative deadlines of tasks are not required to be equal to the period parameters. Another interesting extension comes from considering the results in this paper along-with the work in [4]: given specified affinity masks (as in our model), can we *further constrain* these masks in order to ensure better run-time behavior? Indeed, the approach in [4] can be placed into the following framework:

Given a task set with application- or system-defined initial (maximal) processor affinity sets that the system designer has specified, we perform processor affinity set minimization ***prior to run-time*** and obtain, for each task, the smallest processor affinity set that ensures schedulability. Then ***during runtime*** we perform "global" scheduling that respect the processor affinity sets determined off-line.

Observe that our results in Section V can be considered to fit within this framework: given arbitrary processor affinity sets, the off-line phase would construct, for any feasible system, processor affinity sets in which all but at most m tasks have a processor affinity set of cardinality one, and the sum of the cardinalities of all the processor affinity sets is at most $n+m$ (here, n denotes the number of tasks and m the number of processors).

In this framework, the processor affinity set minimization step can be considered to be an optimization process. It would be interesting to study what kinds of constraints could be placed on this process while retaining tractability, what kinds of constraints are "useful" from a system-implementation perspective, and and what impact such constraints would have on the feasibility/ schedulabilty of the system. As a specific example, how would the optimization algorithm change if the the goal of the optimization step is to obtain a system in which each task may execute on at most two (or in general, some small constant k) processors?

Another interesting generalization of the studied problem was suggested in [4]: processor affinity sets do not put any restrictions on *when* the migrations can take place. It would be interesting to interpret each affinity set as a function of time as well – see [4] for details – in which case the feasibility and run-time scheduling problems become quite a bit more challenging.

Acknowledgements

The work of the first author was supported in part by NSF grants CNS 1016954, CNS 1115284, and CNS 1218693; and ARO grant W911NF-09-1-0535.

References

[1] S. Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th IEEE Real-Time Systems Symposium, RTSS 2004*, pages 37–46, Lisbon, Portugal, December 2004. IEEE Computer Society Press.

[2] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[3] A. Foong, J. Fung, and D. Newell. An in-depth analysis of the impact of processor affinity on network performance. In *Proceedings of the 12th IEEE Conference on Networks, ICON 2004*, pages 244–250, 2004.

[4] A. Gujarati, F. Cerqueira, and B. Brandenburg. Schedulability analysis of the Linux Push and Pull Scheduler with arbitrary processor affinities. In *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, Paris (France), 2013. IEEE Computer Society Press.

[5] P. Hall. On representation of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[6] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[7] L. Khachiyan. A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR*, 244:1093–1096, 1979.

[8] E. Lawler and J. Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM*, 25(4):612–619, Oct. 1978.

[9] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[10] A. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, Washington D.C., May 1988.

[11] G. Nelissen and J. Goossens. A counter-example to: Sticky-ERfair: a task-processor affinity aware proportional fair scheduler. *Real-Time Systems*, 47(4):378–381, July 2011.

[12] V. Salmani, M. Naghibzadeh, and M. Kahani. Deadline scheduling with processor affinity and feasibility check on uniform parallel machines. In *Proceedings of the 7th IEEE International Conference on Computer and Information Technology, CIT 2007*, pages 793–798, 2007.

[13] A. Sarkar, S. Ghose, and P. P. Chakrabarti. Sticky-ERfair: a task-processor affinity aware proportional fair scheduler. *Real-Time Systems*, 47(4):356–377, July 2011.

[14] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1986.

[15] D. B. West. *Introduction to Graph Theory (2nd Edition)*. Prenctice Hall, 2001.