Approximating real-time scheduling on identical machines

Nikhil Bansal[1], Cyriel Rutten[2], Suzanne van der Ster[3], Tjark Vredeveld[2], Ruben van der Zwaan[1]

[1] Eindhoven University of Technology, email: {n.bansal,g.r.j.v.d.zwaan}@tue.nl*
[2] Maastricht University, email: cyrielrutten@gmail.com, t.vredeveld@maastrichtuniversity.nl
[3] Vrije Universiteit, email: suzanne.vander.ster@vu.nl

Abstract. We study the problem of assigning n tasks to m identical parallel machines in the real-time scheduling setting, where each task recurrently releases jobs that must be completed by their deadlines. The goal is to find a partition of the task set over the machines such that each job that is released by a task can meet its deadline. Since this problem is co-NP-hard, the focus is on finding α-approximation algorithms in the resource augmentation setting, i.e., finding a feasible partition on machines running at speed $\alpha \geq 1$, if some feasible partition exists on unit-speed machines.
Recently, Chen and Chakraborty gave a polynomial-time approximation scheme if the ratio of the largest to the smallest relative deadline of the tasks, λ, is bounded by a constant. However, their algorithm has a super-exponential dependence on λ and hence does not extend to larger values of λ. Our main contribution is to design an approximation scheme with a substantially improved running-time dependence on λ. In particular, our algorithm depends exponentially on $\log \lambda$ and hence has quasi-polynomial running time even if λ is polynomially bounded. This improvement is based on exploiting various structural properties of approximate demand bound functions in different ways, which might be of independent interest.

1 Introduction

The *sporadic task system* is one of the most widely adopted models for infinitely recurring executions in real-time systems. Specifically, each sporadic task $\tau = (c_\tau, d_\tau, p_\tau)$ is specified by the amount of processing needed by its jobs c_τ, a deadline d_τ by which a job must be completed, relative to its arrival time, and a minimum interarrival time p_τ between two consecutive jobs, which is called the period of the task. Such a sporadic task releases a possibly infinite sequence of jobs. A sporadic task system $\mathcal{T}$ consists of n sporadic tasks. A task system is said to be *feasible* on a computing platform if for any job sequence that can be possibly generated by the system, there exists a schedule for the task

* Supported by the NWO VIDI grant 639.022.211.

system, such that all jobs from all tasks meet their deadlines. In this paper, we consider the feasibility question of scheduling a set of sporadic tasks to multiple identical machines (processors). This problem and related problems in real-time scheduling have received great attention in the last years; see for example [1, 4, 6] and the references therein.

Single-processor case: Determining the feasibility of a task system on a single (preemptive[4]) processor is quite well-understood. It is well-known that the hardest case for feasibility is when the first jobs of all tasks arrive simultaneously and all subsequent jobs arrive as rapidly as legally possible [5]. That is, we can assume that for each task τ in the task system, the jobs of τ arrive at times $0, p_\tau, 2p_\tau, \ldots$. This sequence of job-arrivals is called the *synchronous arrival sequence*. Another well-known fact [11] is that the Earliest Deadline First (EDF) algorithm, that schedules at any time the job with the earliest absolute deadline, will always produce a valid schedule for any sequence of jobs that is feasible.

Although one can validate whether a task system is feasible by running EDF, this does not provide an efficient polynomial-time feasibility test. The problem is that the periodic nature of jobs together with their relative deadlines can introduce complicated long-range dependencies. In particular, the infeasibility may occur only at a very late time in the schedule, say close to the hyperperiod (which is the least common multiple of the periods of the tasks). In fact, no polynomial-time feasibility test on a uniprocessor is likely to exist, unless P=co-NP [10]. For more results on scheduling sporadic task systems on a single processor, we refer to Baruah and Goosens [4].

Multiprocessor case: For multiprocessor systems, there are two main paradigms for scheduling: *global* vs. *partitioned* scheduling. In partitioned scheduling each task is assigned to one of the machines and all jobs corresponding to this task must be scheduled on that machine. In global scheduling, tasks can use all machines and jobs can even be migrated. Partitioned scheduling is used much more than global scheduling as it is easier to implement and has no communication overhead, which is required if a single task is split between multiple processors. The communication may also lead to security issues. In this paper, we only consider partitioned scheduling.

Observe that in this setting, given a partition of the tasks over the machines, determining its feasibility simply reduces to several independent uniprocessor feasibility problems - one for each machine. Together with the facts for uniprocessor feasibility, the problem we study can be viewed as follows: Find a partition of tasks among machines, such that for each machine, the synchronous arrival sequence for tasks assigned to that machine is feasible for EDF. Clearly, this problem is also co-NP-hard and, as we shall see, it is also NP-hard.

Resource Augmentation and α-feasibility: The hardness of the problem leads us to finding a good approximation algorithm. As usual, we consider the resource

[4] That is, any job can be interrupted arbitrarily during its execution and resumed later from the point of interruption without any penalty. Throughout the paper we consider the preemptive setting.

augmentation setting, where our algorithm is allowed some additional speedup per machine.

Given a parameter $\alpha \geq 1$, we call an algorithm an *α-feasibility test* if it

1. either returns a partition of the tasks into sets $\{\mathcal{T}_i\}_{i\in[m]}$ such that each $\mathcal{T}_i$ can be feasibly scheduled on a machine that runs at speed α; or,
2. returns 'infeasible' if no feasible partition of tasks exists which can be scheduled on m machines running at unit speed.

Alternatively, the algorithm always finds a partition $\mathcal{T}_1, \ldots, \mathcal{T}_m$ of $\mathcal{T}$ such that each $\mathcal{T}_i$ can be feasibly scheduled on a speed-α machine, provided there exists some feasible partition on unit-speed machines.

Let us call a family of feasibility tests a polynomial-time approximation scheme (PTAS), if for any arbitrarily small constant $\epsilon > 0$, there exists a $(1+\epsilon)$-feasibility test in this family with running time polynomial in n and m. Note that the running time dependence on ϵ can be any arbitrary function. If the running time dependence on $1/\epsilon$ is also polynomial, we call the test a fully polynomial-time approximation scheme (FPTAS).

1.1 Related previous results

In the single processor case, an FPTAS feasibility test is known [7]. We will describe a related test later, as its structure will play a key role in our algorithm (see Observation 2 and Theorem 3 below). In particular in this test, one only needs to check the feasibility of the EDF schedule for the job sequence at about $(1/\epsilon)\log(d_{\max}/d_{\min})$ time steps, where $d_{\max}$ and d_{min} are the maximum and minimum task deadlines in the instance.

For partitioned scheduling on multiple machines, Chen and Chakraborty [9] gave a PTAS, generalizing a previous result of [3], if the maximum to minimum deadline ratio is bounded by a constant. Let us call this ratio λ. The idea of [9] is to view the problem as a vector scheduling problem in (roughly) $\ell = (1/\epsilon)\log\lambda$ dimensions. That is, each task is viewed as a ℓ-dimensional vector, and the tasks can be feasibly scheduled on machine, if the corresponding vectors can be feasibly packed in a unit ℓ-dimensional bin. This connection essentially follows from the property for the single-processor test mentioned above. Then the known PTAS for vector scheduling [8] is used in a black-box manner to obtain a $(1+\epsilon)$-approximate feasibility test that runs in time roughly[5] $n^{O(\exp((\frac{1}{\epsilon})\log\lambda))}$. Note that this running time is doubly exponential in $\log\lambda$, and while this is polynomial time for constant λ, it is super-polynomial if λ is super-constant.

If $m = O(1)$, Marchetti-Spaccamela et al. [12] design a PTAS, even for the case that the execution time of a task is machine-dependent.

1.2 Our Contribution

We provide a $(1+\epsilon)$-feasibility test which substantially improves upon the result of Chen and Chakraborty [9]. In particular we show the following result.

[5] For clarity, we suppress some dependence on terms involving $\log\log\lambda$.

Theorem 1. *Given $\epsilon > 0$, a task set $\mathcal{T}$ consisting of n tasks and m parallel identical processors, there is a $(1+\epsilon)$-feasibility test in the partitioned scheduling setting, with running time $O\left(m^{O(f(\epsilon)\log(\lambda))}\right)$. Here $\lambda = d_{\max}/d_{\min}$ and $f(\epsilon) := \exp(O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon})))$ is a function depending solely on ϵ.*

Note that the running time of our algorithm only has a singly exponential dependence on $\log \lambda$, and hence gives an exponential improvement over the result of Chen and Chakraborty [9]. Thus our algorithm can run over a substantially wider range of input instances, beyond just the ones with $\lambda = O(1)$. For example, even if λ is polynomially large in n, our algorithm runs in time $n^{O(\log n)} = 2^{O(\log^2 n)}$ and hence yields a quasi-polynomial-time approximation scheme, as opposed to exponential time by the algorithm in [9].

1.3 High-level Idea

As in Chen and Chakraborty [9], our result is also based on reducing the feasibility problem to vector scheduling in roughly $\ell = (1/\epsilon)\log\lambda$ dimensions. However, we crucially exploit the special structure of the vectors that arise in this transformation and give a faster vector scheduling algorithm for such instances. In fact, as we show in a companion paper [2], exploiting this structure is necessary to obtain any major improvement. In particular, in [2] we show that any PTAS for a general ℓ-dimensional vector scheduling must incur a running time of $\exp((1/\epsilon)^{\Omega(\ell)})$ (under suitable complexity assumptions), and hence the running time in [9] is essentially the best one can hope for if one uses vector scheduling as a black-box.

The starting observation is that even though the vectors corresponding to tasks have $(1/\epsilon)\log\lambda$ coordinates, the number of relevant coordinates are essentially $1/\epsilon$. In particular, only $1/\epsilon$ consecutive coordinates of a vector can have "arbitrary" values, and all subsequent coordinates have an identical value (see (2) and Lemma 2 for the precise statement). This follows from the slack provided by the $1+\epsilon$ speedup, as the demand bound function[6] of a task can be approximated so that a task τ has no "complicating" influence at time points $\geq d_\tau/\epsilon$.

To exploit this structure of vectors, we design a sliding-window based algorithm for vector scheduling, where we carefully build a schedule by considering the coordinates in left to right order, and only keeping track of the relevant short-range information in the dynamic program. The main technical difficulty is to combine the sliding-window approach with the exhaustive enumeration techniques of [8] for vector scheduling. In particular, to ensure that the sliding window does not build up too much error as it moves over the various coordinates, we keep track of different coordinates for a task with different accuracy. Moreover, to keep the running time low, we need more refined enumeration techniques to handle and combine small and large vectors.

[6] The reader unfamiliar with basic concepts of real-time scheduling such as demand bound functions, may wish to first look at Sections 2 and 3 before reading this part.

2 Preliminaries

The input consists of a task system $\mathcal{T}$ consisting of tasks $\tau_1, \ldots, \tau_n$ and a set of m identical processors. Each task releases a sequence of jobs throughout time. Each task τ is characterized by three parameters; the worst-case processing time c_τ, the period p_τ and the relative deadline d_τ. For notational convenience, we write d_j, c_j and p_j instead of d_{τ_j}, c_{τ_j} and p_{τ_j}. Without loss of generality, we assume that all these parameters are integers. The synchronous arrival sequence for $\mathcal{T}$ is defined to be the collection of job arrivals in which each task in $\mathcal{T}$ generates a job at time instant zero and subsequent jobs arrive as soon as permitted by the period parameters, i.e., task τ releases jobs at times $0, p_\tau, 2p_\tau, 3p_\tau, \ldots$ with deadlines $d_\tau, p_\tau + d_\tau, 2p_\tau + d_\tau, 3p_\tau + d_\tau, \ldots$. The utilization of a task τ is $u_\tau := \frac{c_\tau}{p_\tau} \leq 1$, and indicates the share of the processor used by this task in the long run. Without loss of generality, we assume that tasks are ordered such that $d_1 \leq \ldots \leq d_n$. We use $[n] := \{1, 2, \ldots, n\}$ to denote the set of integers from 1 up to n.

In the partitioned scheduling paradigm, we want to find a partition $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$ of $\mathcal{T}$ such that all jobs generated by the tasks in $\mathcal{T}_i$ can be feasibly scheduled on machine i, for all $i \in [m]$. Furthermore, task set $\mathcal{T}_i$ can be feasibly scheduled on machine i if the synchronous arrival sequence for tasks in $\mathcal{T}_i$ can be scheduled feasibly by the Earliest Deadline First (EDF) algorithm. This implies that a task set $\mathcal{T}_i$ can be feasibly scheduled on the machine if and only if the total workload of the jobs generated by tasks in $\mathcal{T}_i$ that need to be finished by time t is not more than the amount of work machine i can do up to time t.

Demand bound function: It is known [5] that task system $\mathcal{T}_i$ is feasible upon a preemptive uniprocessor if and only if

$$\sum_{\tau \in \mathcal{T}_i} \max\left\{0, \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_\tau \right\} \leq t, \qquad \forall t > 0. \tag{1}$$

The term $\max\left\{0, \left\lfloor \frac{t+p_\tau-d_\tau}{p_\tau} \right\rfloor c_\tau \right\}$ is known as the *demand bound function* (dbf) of task τ at time point t and is denoted by $\mathrm{dbf}_\tau(t)$. It expresses the amount of processing task τ needs up to time t. The left-hand side of (1) is called the dbf for the task set $\mathcal{T}_i$ and denoted by $\mathrm{dbf}_{\mathcal{T}_i}(t)$.

Condition (1) can be weakened slightly, and it is easy to see that it suffices to check (1) only for times t that are deadlines of some job and $t \leq p_{lcm}$ where p_{lcm} denotes the least common multiple of the tasks' periods. However, given that the feasibility testing problem is co-NP-hard, it is unlikely that the number of points where (1) must be tested can be reduced substantially.

As mentioned above, our goal is to develop a $(1+\epsilon)$-feasibility test for any $\epsilon > 0$. As we shall see soon, if we only care about $(1+\epsilon)$-feasibility, it suffices to check condition (1) at only $\log_{(1+\epsilon)}(d_n/(\epsilon^2 d_1)) \approx O(\log(d_n/d_1)/\epsilon)$ time points. This allows us to transform the feasibility problem into the so-called vector scheduling problem, which is defined as follows.

Definition 1 (Vector Scheduling). *We are given a set A consisting of n d-dimensional vectors with each coordinate in the range $[0, 1]$ (i.e., vectors in*

$[0,1]^d$*), and a positive integer* m*. The goal is to determine whether there is a partition of* A *into* m *sets* $A_1, \ldots, A_m$ *such that for each set* A_i*, the sum of vectors in that set does not exceed* 1 *in any coordinate. That is,* $\max_{1 \le i \le m} \left\| \sum_{a \in A_i} a \right\|_\infty \le 1$.

Chekuri and Khanna [8] showed the following result for vector scheduling.

Theorem 2 ([8]). *Given any* $\epsilon > 0$*, there is a* $(1+\epsilon)$*-approximation algorithm, i.e., an algorithm that finds a partition with* $\max_{1 \le i \le m} \left\| \sum_{a \in A_i} a \right\|_\infty \le 1 + \epsilon$*, for the vector scheduling problem that runs in time* $n^{O(s)}$*, where* $s = (\frac{\ln d}{\epsilon})^d$.

In the following section, we show how $(1+\epsilon)$-feasibility reduces to vector scheduling with $d = O((1/\epsilon)\log(d_n/d_1))$. While similar results have been used before (e.g., [9, 12]), we will repeat the proof here, as we explicitly need the structure of the vectors in the resulting vector scheduling instance, which our algorithm will crucially exploit later.

3 From Sporadic Task System to Vector Scheduling

We begin with the notion of *approximate* demand bound functions. Observe that over the long run, a task τ uses c_τ units of time every p_τ units of time, but the relative deadlines, that may be different from the periods, complicate the demand bound function. The demand bound function has sharp jumps at the (absolute) deadlines $d_\tau, p_\tau + d_\tau, 2p_\tau + d_\tau, \ldots$, but the effects of these jumps become milder as time progresses. A machine that is $1+\epsilon$ times faster gives ϵt units of extra processing time up to time t, which lets us ignore these sharp jumps after a certain point in time and instead it is sufficient to use the *utilization* (the average processing requirement).

The next lemma shows that we only need to check the demand bound function at time points which are a factor $1+\epsilon$ apart.

Lemma 1. *For any task* τ*, if* $dbf_\tau((1+\epsilon)^k d_1) \le (1+\epsilon)^k d_1 \alpha$ *for all* $k \in \mathbb{N}_{\ge 0}$*, then* $dbf_\tau(t) \le (1+\epsilon)\alpha t$ *for all* $t \ge 0$.

Proof. For any t, define integer k_t such that $(1+\epsilon)^{k_t - 1} d_1 < t \le (1+\epsilon)^{k_t} d_1$. Then

$$\mathrm{dbf}_\tau(t) \le \mathrm{dbf}_\tau((1+\epsilon)^{k_t} d_1) \le (1+\epsilon)^{k_t} d_1 \alpha < (1+\epsilon)\alpha t,$$

where the first inequality follows from the demand bound function being non-decreasing. □

We consider an approximate demand bound function $\mathrm{dbf}^*_\tau(t)$ used by Marchetti-Spaccamela et al. [12]. Let L be the smallest integer such that $1 \le (1+\epsilon)^{L-1}\epsilon^2$. Note that $L \le 2 + \log_{(1+\epsilon)}(1/\epsilon^2)$. Let

$$\mathrm{dbf}^*_\tau(t) = \begin{cases} \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_\tau & \text{if } t < (1+\epsilon)^L d_\tau, \\ u_\tau t & \text{otherwise.} \end{cases} \tag{2}$$

Note that dbf^* differs from dbf only when $t \geq d_\tau(1+\epsilon)/\epsilon^2$, and is proportional to the utilization of τ in that case. The following lemma shows that it is a good approximation to dbf.

Lemma 2 ([12]). *For every task τ and every time $t \geq 0$,*

$$\frac{1}{(1+\epsilon)}\, dbf_\tau(t) \leq dbf^*_\tau(t) \leq (1+\epsilon)\, dbf_\tau(t).$$

Another obvious property of dbf and dbf^* is the following, which allows us to start our feasibility analysis at the first deadline only.

Observation 1 *For all tasks τ, for all $t < d_\tau$, we have that*

$$dbf_\tau(t) = dbf^*_\tau(t) = \max\{0, \lfloor (t + p_\tau - d_\tau)/p_\tau \rfloor\} = 0.$$

In particular, $dbf_\tau(t) = dbf^*_\tau(t) = 0$ for all $t < d_1$ and all tasks $\tau \in \mathcal{T}$.

Using Lemma 1, Lemma 2 and Observation 1, we can encode our approximate demand bound function dbf^*_τ into a vector w^τ. More precisely, we will use a *normalized* demand bound function which is $\mathrm{dbf}^*_\tau(t)/t$. Let $t_{end} = (1+\epsilon)^L d_n$, and let $K := \lceil \log_{(1+\epsilon)} t_{end}/d_1 \rceil$. For each task τ we define the vector w^τ, with coordinates w^τ_k as follows:

$$w^\tau_k := \begin{cases} \frac{\mathrm{dbf}^*_\tau\left((1+\epsilon)^{k-1} d_1\right)}{(1+\epsilon)^{k-1} d_1} & \text{if } k = 1, \ldots, K-1, \\ u_\tau & \text{if } k = K. \end{cases}$$

Note that the first $K-1$ coordinates of these vectors consider times that are powers of $(1+\epsilon)$ and lie between d_1 and $(1+\epsilon)^L d_n$. Recall that for $t \geq t_{end}$, it holds that $\mathrm{dbf}^*_\tau(t) = u_\tau t$ for each task $\tau_1, \ldots, \tau_n$. Thus, there is no need to consider additional coordinates. The coordinate K is equal to the utilization and will play a special role in our algorithm.

We note the following structural property of the vectors w_τ.

Observation 2 *A task τ is associated to a vector w^τ from $[0,1]^K$ with $K := 1 + \lceil \log_{(1+\epsilon)} \frac{d_n}{\epsilon^2 d_1} \rceil$:*

$$w^\tau_k := \begin{cases} 0 & \text{if } k \leq k_\tau, \\ \frac{dbf^*_\tau(t_k)}{t_k} & \text{if } k = k_\tau + 1, \ldots, k_\tau + L, \\ u_\tau & \text{otherwise,} \end{cases} \tag{3}$$

where $t_k = (1+\epsilon)^k d_1$ and $k_\tau = \lceil \log_{(1+\epsilon)}(d_\tau/d_1) \rceil - 1$.

In particular, each vector has initial coordinates zero, followed by at most L entries of arbitrary value, followed by all entries equal to u_τ.

The following theorem connects the vector scheduling problem formally to the sporadic task system scheduling, and follows directly from Lemmas 1 and 2, (1) and Observation 1.

Theorem 3. *Define the vectors w^τ as in* (3)*. Given is a partition of vectors w^τ into m sets $W_1, \ldots, W_m$ and the corresponding partition of tasks $\tau \in \mathcal{T}$ into m sets $\mathcal{T}_1, \ldots, \mathcal{T}_m$. Then, for all machines i,*

(i) if $\left\|\sum_{w^\tau \in W_i} w^\tau\right\|_\infty \leq \alpha$, then $dbf_{\mathcal{T}_i}(t) \leq (1+\epsilon)^2 \alpha t$ for all $t \geq 0$;
(ii) if $dbf_{\mathcal{T}_i}(t) \leq \alpha t$ for all $t \geq 0$, then $\left\|\sum_{w^\tau \in W_i} w^\tau\right\|_\infty \leq (1+\epsilon)\alpha$.

This theorem tells us that if we can partition the set of vectors $w^\tau \in W$ into sets $W_1, \ldots W_m$ such that $\| \sum_{w^\tau \in W_i} w^\tau \|_\infty \leq 1+\epsilon$, for all $i \in [m]$, then we can feasibly schedule the corresponding tasks in set $\mathcal{T}_i$ on machine i if this machine receives a speedup factor $(1+\epsilon)^3$. Moreover, if $\mathcal{T}$ can be partitioned into sets $\mathcal{T}_i$ such that each of these can be scheduled on a unit-speed machine, then the corresponding sets of vectors W_i satisfy $\| \sum_{w^\tau \in W_i} w^\tau \|_\infty \leq 1+\epsilon$, for all $i \in [m]$.

Thus, a $(1+\epsilon)$-approximation for vector scheduling implies a $(1+\epsilon)^2(1+\epsilon) = 1 + O(\epsilon)$-feasibility test for partitioned scheduling. The result of Chen and Chakraborty [9] follows directly from this connection, and applying Theorem 2. In the next section we show how the running time can be improved for vector scheduling by exploiting the special structure of the vectors w^τ as described in Observation 2.

4 Solving the special case Vector Scheduling problem

In this section we develop a substantially faster $(1+\epsilon)$-approximation algorithm for vector scheduling, which is specifically tailored towards vectors described in Observation 2. We combine several techniques from bin packing and vector scheduling and design a "sliding window" dynamic programming approach. The time complexity of our algorithm is given in the following theorem. Note that this theorem and Theorem 3 of the previous section suffice to prove Theorem 5.

Theorem 4. *Given $\epsilon > 0$, let $C = \left(\left\lceil \frac{8L+19}{\epsilon} \right\rceil\right)^L \left\lceil \frac{K(8L+19)}{\epsilon} \right\rceil$ where $L = 1 + \lceil \log_{(1+\epsilon)}(1/\epsilon^2) \rceil$ and $K = 1 + \lceil \log_{(1+\epsilon)}(d_n/\epsilon^2 d_1) \rceil$. Then, given a set of vectors W from $[0,1]^K$ as defined in Observation 2, Algorithm 1 determines in $O(m^{O(C)})$ time whether the set of vectors W can be scheduled on m machines such that in every coordinate the load is at most $1+\epsilon$, or whether no feasible assignment exists.*

Proof. The theorem follows easily from Lemma 3 which will follow in Section 4.3. Setting $\eta := \epsilon/(8L+19)$ in Lemma 3 leads to a schedule with height at most $1 + (8L+19)\eta = 1+\epsilon$. □

The main idea of the algorithm is, after some rounding of the vectors, to first classify the vectors, then determine how the vectors of one class can possibly be scheduled and finally to combine the schedule of the classes into one overall schedule. To give a high-level overview of our algorithm in Section 4.2 and some of the details in the subsequent subsection, we first need to introduce some notation and concepts in the following subsection.

4.1 Notation and definitions.

Given $\epsilon > 0$, let L and K be defined as above. Let $0 < \eta < 1$ be a small constant and define $\delta = \eta/K$.

We associate each task τ to a vector w^τ from $[0,1]^K$ as defined in (3). We classify these vectors into several classes depending on the index of the first non-zero coordinate. Hereto, we say that a vector is a **t-vector** if its first non-zero coordinate is coordinate t.

A **t-configuration** is an $(L+1)$-tuple $(f_1, \ldots, f_L, f_u)$ with, for all $k \in [L]$, $f_k \in \{0, \eta, 2\eta, ..., \eta\lfloor 1/\eta \rfloor, 1\}$ and $f_u \in \{0, \delta, 2\delta, \ldots, \delta\lfloor 1/\delta \rfloor, 1\}$. We say that (a set of vectors assigned to) a machine i **conforms to** a t-configuration $f = (f_1, \ldots, f_L, f_u)$ if the contribution to coordinate $t-1+k$ is at most f_k, for all $k \in [L]$, and if the contribution to all coordinates $k \geq t+L$ is at most f_u. As the first L elements in a t-configuration can attain one of $\lceil 1/\eta \rceil$ different values and the last element can attain one of $\lceil 1/\delta \rceil$ different values, the number of different t-configurations, denoted by C, is $C := \left(\left\lceil \frac{1}{\eta} \right\rceil\right)^L \left\lceil \frac{1}{\delta} \right\rceil$.

A **t-profile** Q defines a t-configuration for each machine. Therefore, it can be represented by an m-tuple $Q = (q_1, \ldots, q_m)$ where q_i denotes the t-configuration corresponding to machine i. On the other hand, as the number of t-configurations is bounded by C and the machines are identical, a t-profile can also be represented by a C-tuple $Q = \langle n_1, \ldots, n_C \rangle$ where n_f denotes the number of machines that conform to configuration f. As the numbers n_f sum up to m, we find that the number of different t-profiles is at most m^C.

Finally, we define the addition of a t-profile Q and a vector $e = (e_1, \ldots, e_L, e_u) \in [0,1]^{L+1}$, $Q + e = Q' = (q'_1, \ldots, q'_m)$, as the pointwise addition of the vector e to each configuration $q_i \in Q$, i.e., $q'_i = q_i + e$ for all $i \in [m]$.

4.2 Overview of the algorithm.

Our algorithm, given in Algorithm 1, determines whether we can feasibly schedule all vectors with a load of at most $1+\epsilon$ in every coordinate on each machine. It first applies two rounding steps (see Step 2 and 3), to limit the number of different vectors.

In Step 4 of the algorithm, we determine for each $t = 1, \ldots, K$ and t-profile R whether all t-vectors can be scheduled to conform to R. Due to lack of space the proof of this is omitted.

Once we know, for every t, conforming to which t-profiles the set of all t-vectors can be scheduled, we can determine conforming to which t-profiles all vectors together can be scheduled. Hereto, we design a *sliding window DP* to determine whether all k-vectors ($k < t$) can be combined with all t-vectors to conform to a given t-profile Q (Section 4.3). The final result can then easily be obtained by taking $t = K$ and Q equal to the all-1 profile, i.e., $q_i = \mathbf{1}$ for all i. When $T[K, \mathbf{1}]$ returns true, Algorithm 1 also can be used to find the corresponding solution.

Both Step 4 and Step 5 of Algorithm 1 need to be able to determine whether a t-profile R and a $(t-1)$-profile (or t-profile) S can be combined into a t-profile Q. This can be determined in advance in $O(m^{O(C)})$ time, but the proof is omitted.

Algorithm 1 Vector Scheduling algorithm

Require: Input: a set W of vectors w^τ as defined in Section 3, and $\eta > 0$.

1: Define $\delta := \eta/K$.
2: For each vector w^τ round each component w_k^τ down to the nearest power of $\frac{1}{1+\eta}$.
3: Modify each vector

$$z_k^\tau := \begin{cases} 0 & \text{if } w_k^\tau \leq \delta \left\| w^\tau \right\|_\infty, \\ w_k^\tau & \text{otherwise.} \end{cases}$$

4: Determine whether all t-vectors can be scheduled conforming to t-profile R, for all possible t-profiles R and all t.
5: Let $T[t, Q]$ be true if all k-vectors with $k \leq t$ can be scheduled conforming to t-profile Q, and false otherwise. Determine $T[t, Q]$ for all possible t-profiles Q and all t.
6: Return $T[K, 1]$.

4.3 The Sliding Window Dynamic Program

In this subsection, we introduce a dynamic program to determine whether all k-vectors with $k \leq t$ can be scheduled conforming to t-profile Q. To be precise, we compute the values $T[t, Q]$, which essentially evaluates to true if all k-vectors with $k \leq t$ can be scheduled conforming to t-profile Q. The dynamic program works in K phases as it moves from the first coordinate to coordinate K. While scheduling all t-vectors in a certain phase t, the DP also looks ahead to the next $L-1$ coordinates and the last utilization coordinate to ensure no conflicts arise in these coordinates. That is, we slide a window covering L coordinates from coordinate 1 to coordinate K in as many phases.

Intuitively, phase t corresponds to scheduling the t-vectors, given a partial schedule for all k-vectors with $k < t$. To determine the value of $T[t, Q]$, we split the t-profile Q into a t-profile R and a $(t-1)$-profile S that capture the division of space per machine and per coordinate between the t-vectors and the other k-vectors with $k < t$.

Since the t-configurations are "coarse valued" (all values are multiples of either η or δ), it is unclear how to split the t-profile Q: perhaps a coordinate f_k of the t-configuration can be split into two parts yielding a feasible t-profile R and a $(t-1)$-profile S, but not in such a way that the two parts are multiples of η. In that case, the corresponding DP-cell is erroneously evaluated to false. To circumvent this issue, an additional small error in each phase of the sliding window DP is allowed. For this reason the vector $(\eta, \ldots, \eta, \delta)$ is added to the $(t-1)$-profile S.

The boolean value $B[t, R]$, that essentially denotes whether all t-vectors can be scheduled conforming to t-profile R, can be precomputed in $O(m^{O(C)})$ time (proof is omitted). Once these values are known, the recursive formula for T can be easily computed by considering all possible combinations of t-profiles R and $(t-1)$-profiles S that can be combined to a t-profile Q, and determining whether or not all t-vectors can be scheduled conforming to R and all other k-vectors with $k < t$ can be scheduled conforming to $S + (\eta, \ldots, \eta, \delta)$. That is, for $t > 1$,

$$T[t, Q] = \bigvee_{(R,S)\in\mathcal{W}(Q)} (B[t, R] \wedge T[t-1, S + (\eta, \ldots, \eta, \delta)]), \tag{4}$$

where $\mathcal{W}(Q)$ contains all tuples (R, S) of t-profiles R and $(t-1)$-profiles S that Q can be split into. The base case of the recursion is

$$T[1, Q] = B[1, Q]. \tag{5}$$

To evaluate the running time of computing $T[K, Q]$, we note that $B[t, R]$ is precomputed and can be accessed in $O(1)$ time. The proof of the following lemma is omitted due to lack of space.

Lemma 3. *Let W be a set of vectors w^τ as defined in* (3) *and let $\eta > 0$ be small enough. Algorithm 1 decides in $O(Km^{O(C)})$ time whether there exists a partition of the vectors W into m sets $W_1, \ldots, W_m$ such that $\left\|\sum_{w^\tau\in W_i} w^\tau\right\|_\infty < 1+(8L+19)\eta$ for all i, or that there does not exist a partition with $\left\|\sum_{w^\tau\in W_i} w^\tau\right\|_\infty \leq 1$.*

Note that if we choose $\eta = \epsilon/(8L + 19)$, we prove Theorem 4 and find a partition of height at most $1 + \epsilon$.

5 Conclusion

Combining Theorem 3 and Theorem 4 yields the desired result.

Theorem 5. *Given $\epsilon > 0$, a task set $\mathcal{T}$ and m parallel identical processors, there is an algorithm which correctly decides in $O\left(m^{O(f(\epsilon)\log\lambda)}\right)$ time whether $\mathcal{T}$ can be feasibly partitioned with a speedup of $1 + \epsilon$, or no feasible partition exists in case the machines run at unit speed, where $\lambda = d_n/d_1$, the ratio between the largest and smallest deadline, and $f(\epsilon)$ is a function depending solely on ϵ.*

Proof. Theorem 4 determines in $O(m^{O(C)})$ time whether a feasible solution to the vector scheduling problem exists with a speedup factor of $1 + \epsilon$, or whether no such partition of the vectors to the machines exists without speedup. Thus in light of Theorem 4, Theorem 3 implies that if there exists a feasible partition for the vector scheduling problem, then this partition is feasible for our real-time scheduling problem if the machines receive a speedup factor of $(1+\epsilon)^3$, and that if no feasible partition for the vector scheduling problem exists, then no feasible partition exists for the real-time scheduling problem in case the machines run at speed $1/(1 + \epsilon)$. Rescaling ϵ appropriately yields the stated result.

References

1. T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*, chapter 3. CRC Press, 2007.
2. N. Bansal, T. Vredeveld, and R. van der Zwaan. Approximating vector scheduling: almost matching upper and lower bounds. Submitted to LATIN 2014.
3. S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of 26th IEEE Real-Time Systems Symposium*, pages 321–329. IEEE, 2005.
4. S. Baruah and J. Goossens. Scheduling real-time tasks: Algorithms and complexity. In J.Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models and Performance Evalution*, chapter 28. CRC Press, 2004.
5. S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of 11th IEEE Real-Time Systems Symposium*, pages 182–190. IEEE, 1990.
6. S. K. Baruah and K. Pruhs. Open problems in real-time scheduling. *Journal of Scheduling*, 13:577–582, 2010.
7. S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *Proceedings of 23rd IEEE Real-Time Systems Symposium*, pages 159–168. IEEE, 2002.
8. C. Chekuri and S. Khanna. On multi-dimensional packing problems. *SIAM Journal on Computing*, 33(4):837–851, 2004.
9. J.-J. Chen and S. Chakraborty. Partitioned packing and scheduling for sporadic real-time tasks in identical multiprocessor systems. In *Proceedings of 24th Euromicro Conference on Real-Time Systems*, pages 24–33, 2012.
10. F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of 21st Symposium on Discrete Algorithms*, pages 1029–1034, 2010.
11. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.
12. A. Marchetti-Spaccamela, C. Rutten, S. van der Ster, and A. Wiese. Assigning sporadic tasks to unrelated parallel machines. In *Proceedings of 39th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 665–676, 2012.