Partitioned EDF scheduling on a few types of unrelated multiprocessors[*]

Andreas Wiese Vincenzo Bonifaci Sanjoy Baruah

Abstract

A polynomial-time approximation scheme (PTAS) is derived for the partitioned EDF scheduling of implicit-deadline sporadic task systems upon unrelated multiprocessor platforms that are comprised of a constant number of distinct types of processors. This generalizes earlier results showing the existence of polynomial-time approximation schemes for the partitioned EDF scheduling of implicit-deadline sporadic task systems on (1) identical multiprocessor platforms, and (2) unrelated multiprocessor platforms containing a constant number of processors.

1 Introduction

Embedded and real-time systems are increasingly coming to be implemented upon multicore platforms. As the computational demands made by ever more complex applications continue to increase, there is a need for enhanced performance capabilities from these platforms. Initially, the approach adopted for obtaining such enhanced performance was to increase core counts. Soon, however, chip makers began to distinguish themselves not by offering more general-purpose cores, but by providing specialized hardware components that accelerate particular computations. Examples include multicore CPUs with specialized graphics processing cores, specialized signal-processing cores, specialized floating-point units, customizable FPGAs, etc. Computing platforms such as these with specialized components are called *heterogeneous* or *unrelated* multiprocessor platforms.

We consider here the partitioned preemptive EDF scheduling of implicit-deadline sporadic task systems (also known as *Liu & Layland (LL)* task systems [11]) on such unrelated multiprocessor platforms. This is a difficult problem – even on simpler *identical* multiprocessor platforms (in which all the processors are assumed to have the same capabilities), it is known (see, e.g., [12]) that such partitioning is highly intractable: NP-hard in the strong sense. Optimal partitioning strategies are therefore unlikely to have polynomial-time implementations. Techniques are known (see, e.g., [2]) for representing this partitioning problem as an integer linear program (ILP). Although solving an ILP exactly takes exponential time, approximation techniques [10] can be used to solve these ILPs *approximately*, in polynomial time. These approximation

[*]This was LaTeXed on February 13, 2012.

algorithms make the following worst-case guarantee: any task system that can be partitioned by an optimal algorithm on a given heterogeneous multiprocessor platform can be partitioned by these algorithms upon a platform in which each processor is twice as fast.

PTAS's. A Polynomial-Time Approximation Scheme (PTAS) is an algorithm for solving certain kinds of optimization problems approximately. A PTAS for a given optimization problem takes as input a parameter $\delta > 0$ and an instance of the optimization problem and, in time polynomial in the problem size (although not necessarily in the value of δ), produces a solution that is within a factor $(1 + \delta)$ of being optimal.

For the case of identical multiprocessors, results by Hochbaum and Shmoys [6] can be directly applied to obtain a PTAS for partitioning LL systems upon a given multiprocessor platform, which makes the following guarantee: Given a $\delta > 0$, an LL task system, and an identical multiprocessor platform, if an optimal partitioning algorithm can partition the task system upon the platform then the PTAS in [6] can partition the same task system upon a platform containing the same number of processors, each of which is $(1 + \delta)$ times as fast.[1] This result was later generalized [7] to *uniform* multiprocessors, which are multiprocessor platforms in which each processor may have a different speed but executing different tasks on a faster processor speeds up their executions by the same constant multiplicative factor. However, it is known that no such PTAS can exist for unrelated multiprocessors unless P = NP. In fact, [10] has shown that under the assumption that P $\neq$ NP, no polynomial-time algorithm with an approximation ratio smaller than $3/2$ can exist for this problem. To our knowledge, the approximation ratio from [10] of two, mentioned above, is the best approximation result known for heterogeneous multiprocessors.

Limited unrelated multiprocessors. The lower bound of $3/2$ from [10] provides strong evidence that we are unlikely to be able to approximately partition LL tasks on unrelated multiprocessors to an arbitrary constant degree of accuracy, in polynomial time. However, Andersson, Raravi and Bletsas [1] argue persuasively in a recent paper that it is interesting to study heterogeneous multiprocessor platforms in which the number of distinct *types* of processors is a (relatively small) constant. They point out the plethora of currently-available and soon-to-be-available multicore CPUs with two distinct types of processors — typically one or a few general-purpose processing cores and one or more specialized graphics processors, or general-purpose processing cores and "synergistic" processors for executing SIMD instructions. Such example multicore CPUs provide motivation to consider the problem of partitioning LL task systems upon unrelated multiprocessors in which all the processors are of a relatively small number of distinct types. We refer to such multiprocessors as *limited unrelated multiprocessors*.

[1] Equivalently, if the PTAS fails to partition the task system upon a given identical multiprocessor platform then no algorithm can partition it upon a platform of the same number of processors, each $1/(1 + \delta)$ times as fast.

This research. Given this increasing industrial trend towards CPUs that can be modeled as limited unrelated multiprocessor platforms, it is interesting to ask whether one can exploit specific properties of such platforms to design polynomial-time partitioning algorithms that have a better worst-case approximation ratio than the currently smallest known value of two (or indeed, better than the lower bound of $3/2$) for the general problem? In this paper, we answer this question in the affirmative, by demonstrating the existence of a PTAS for partitioning LL task systems on platforms with a constant number of distinct processor types.

A note. The PTAS we present in this paper has been designed to demonstrate that partitioning algorithms of polynomial-time complexity exist, that offer arbitrarily good approximations to optimality. We have not attempted to make this PTAS exhibit good run-time performance "in practice." For example, for ease of explanation we have at several points chosen to perform certain steps in a computationally more expensive (although still polynomial-time) manner than is needed. We hope that our result here will encourage research into seeking out polynomial-time partitioning algorithms that exhibit better run-time behavior in practice, and perhaps give rise to look-up table (LUT) based scheduling strategies (see, e.g., [4], where the theoretically significant but too-slow-to-implement in practice PTAS for partitioning on identical multiprocessors was rendered practically useful by means of a LUT).

2 Task and platform model

In a *limited unrelated multiprocessor* platform, there are κ distinct types of processor for some constant κ, and all the processors in the platform are restricted to being of one of these κ types.

An implicit-deadline sporadic task (LL task) is characterized by its worst-case execution times on each type of processor, and its period [11]. Since we are concerned with partitioned EDF scheduling on preemptive platforms, however, it follows from the well-known preemptive uniprocessor utilization-based feasibility test for LL task systems [11] that the parameters of significance are the *utilizations* of the task on each type of processor. We are given a task system of n Liu and Layland tasks to be partitioned upon an m-processor limited unrelated multiprocessor platform with κ distinct types of processors. For each $i, 1 \leq i \leq n$, the i'th task is specified by specifying the m utilization values u_{ij}, $1 \leq j \leq m$. The interpretation is that the utilization of the i'th task if implemented on the j'th processor is equal to u_{ij}. Note that these u_{ij} values must be consistent with the processor types: if the j'th and j''th processors are of the same type then $u_{ij} = u_{ij'}$ for each i, $1 \leq i \leq n$. Given such specifications, we seek a feasible partitioning of the task system upon the multiprocessor platform, where a *feasible* partitioning is defined as follows:

Definition 1 (feasible partitioning) *A feasible partitioning of a system of Liu and Layland tasks upon a multiprocessor platform is a partitioning of the tasks among the processors in which the total utilization assigned to each processor does not exceed*

n	number of tasks
κ	number of different types of processors
m	number of processors
u_{ij}	the utilization of the i'th task, if implemented on the j'th processor
ϵ	desired degree of accuracy
E	number of integer powers of $(1+\epsilon)$ in $(\epsilon, 1]$ (Eqn 1)
big tasks	the i'th task is *big* for the j'th processor if $u_{ij} > \epsilon$. (Definition 2)
small tasks	the i'th task is *small* for the j'th processor if $u_{ij} \leq \epsilon$. (Definition 2)

Table 1: Terminology and notation

the capacity of the processor. That is, for each processor j it is the case that

$$\sum_{\textit{(the i'th task is assigned to the j'th processor)}} u_{ij} \leq 1$$

■

Let ϵ denote a constant > 0; informally, this represents the degree of accuracy we desire in our PTAS.

3 Background

In this section we briefly review some concepts and results that we will use in deriving our PTAS.

3.1 Linear and Integer Linear programs

In a Linear Program (LP) over a given set of variables, one is given a collection of "constraints" that are expressed as linear inequalities over these variables, and optionally an "objective function," also expressed as a linear inequality of these variables. If an objective function is specified then the goal is to find some assignment of values to the variables satisfying all the constraints, that results in an extremal (maximum/ minimum) value of the objective function; if no objective function is specified then the goal is to determine some assignment of values to the variables that satisfies all the constraints.

It is known that an LP can be solved in time polynomial in its representation by the ellipsoid algorithm [9] or the interior point algorithm [8]. (In addition, the exponential-time simplex algorithm [5] has been shown to perform extremely well "in practice," and is often the algorithm of choice despite its exponential worst-case behaviour.) We do not need to understand the details of these algorithms: for our purposes, it suffices to know that LP problems can be efficiently solved (in polynomial time).

An Integer Linear Program (ILP) is a linear program with the additional constraint that some or all of the variables are restricted to take on integer values only. Differently from LPs, solving ILPs is known to be intractable (NP-hard in the strong sense [13]).

The Linear Program obtained from any Integer Linear Program by relaxing the requirement that the variables take on integers values only, is referred to as the *LP-relaxation* of the ILP.

3.2 Matchings on Bipartite graphs

A *bipartite graph* $G = (V, W, E)$ is specified by specifying two disjoint sets of nodes V and W and a set of edges E where every edge in E has one endpoint in each of V and W. We say that $E' \subset E$ is a *matching* for V in this graph if

1. for each node $v \in V$ there is exactly one edge of E' incident to v, and

2. for each node $w \in W$ there is at most one edge of E' incident to w.

(We assume that $|V| \leq |W|$; otherwise, no matching of V exists.)

A *fractional matching* is a generalization of a matching: a fractional matching for V is an assignment of non-negative *weights* on the edges in E such that (i) For each $v \in V$, the weights of all the edges incident on v sum to exactly one; and (ii) For each $w \in W$, the weights of all the edges incident on w sum to at most one. A "normal" matching (often called an *integral matching*) can thus be thought of as a fractional matching satisfying the additional condition that each edge weight is either zero or one.

There is an algorithm due to Birkhoff [3] that, *given any fractional matching for a bipartite graph, determines an integral matching in polynomial time* — see [14] for details. We will use this result later; hence we formally state it as a theorem here.

Theorem 1 (Birkhoff's algorithm) *Given any fractional matching for a bipartite graph, it is possible to determine an integral matching on the same graph in time polynomial in the representation of the graph.*

4 Our partitioning strategy: outline

Given a collection of n implicit-deadline sporadic tasks to be partitioned on an m-processor unrelated multiprocessor platform consisting of κ different types of processors, this is how we seek to obtain a feasible partitioning.

- First, we transform the task system into another that is easier to partition. We will show that the transformed system can be partitioned upon the given platform provided the original system can be partitioned upon a platform in which each processor is slower by a factor $1/(1 + \epsilon)$.

  This transformation process is described in Section 5. Once we have completed the transformation, we deal exclusively with this transformed task system.

- Next, we consider the platform. In Section 6 we distinguish between *big* and *small* tasks, and introduce the notion of *patterns of big tasks*. We will see that patterns of big tasks are a characterization of any feasible (i.e., successful) partitioning of any task system on the platform under consideration, in the sense that for any feasible partitioning one can define a pattern of big tasks characterizing that partitioning. We will show that upon the given platform, the number of different patterns of big tasks that characterize feasible partitionings is bounded from above by a polynomial in the number of processors.

- Since there are only polynomially many different patterns of big tasks that characterize feasible partitionings, it is possible, in polynomial time, to spend a polynomial amount of time examining each of these patterns. That, therefore, is what we do in Section 7. For each pattern,

    1. We assume that it corresponds to a feasible partitioning of the task system that we wish to partition, upon a platform in which each processor is slower by a factor $(1 - \epsilon)$.
    2. Under this assumption, we construct an ILP that represents the feasible partitioning, with the integer variables in the ILP denoting where each task gets allocated. This is done in Section 7.1.
    3. Solving an ILP is an NP-hard problem. However if the ILP is "relaxed" by removing the restriction that the variables take on integer values, then an assignment of values to the variables that satisfies all the constraints can be determined in polynomial time, if such an assignment of values exists. If no such assignment of values exists, we move on to the next pattern; if we have exhausted all the patterns then we stop and declare failure: we are unable to find a feasible partitioning. We will see that such failure implies that there is *no* feasible partitioning of the task system (on a platform in which each processor is slower by a factor $(1 - \epsilon)$).

- If we did not report failure above, we will have obtained a (fractional) solution to the relaxation of the ILP. This fractional solution may allocate a single task across multiple processors (and therefore does not represent a feasible partitioning of the task system). We obtain a feasible partitioning from this fractional solution in the following manner.

    1. We construct a bipartite graph from the fractional solution, and define a fractional matching on this graph that corresponds to the fractional allocation of the tasks on the processors. We describe how this graph and the matching are determined, in Section 7.2.
    2. We then use the algorithm of [3] (mentioned in Theorem 1 above) to determine an integer matching from the fractional one.
    3. Finally in Section 7.3, we use the integer matching to determine a partitioning of the task system. Recall that we had constructed our ILP under the assumption that the pattern of big tasks under consideration corresponds to a feasible partitioning of the task system upon a platform in which each processor was slower by a factor $(1 - \epsilon)$. Given this assumption, we show that the partitioning we have obtained is feasible upon the given platform.

Note that there are two speedup factors involved: a $\frac{1}{1+\epsilon}$ factor in the initial transformation of the task system, and a $(1 - \epsilon)$ factor in the construction of the ILP and the consequent determination of the feasible partitioning. We will therefore have shown that *if our algorithm fails to partition a given task system upon a specified platform, then no algorithm can partition the task system upon a platform in which each processor is slower by a factor* $\left(\frac{1-\epsilon}{1+\epsilon}\right)$.

5 Initial transformation of the task system

In this section we describe how we transform the task system to one that becomes easier to partition, and show (Lemma 1) that given any partitioning of the transformed system upon a platform we can obtain a partitioning of the original system upon a platform in which each processor is faster by a factor $(1+\epsilon)$.

The transformation consists of *rounding up* each u_{ij} value to the largest integer power of $(1+\epsilon)$.

For example, if $\epsilon = 0.25$ then the successive integer powers of ϵ are $1.25^0 = 1.0$; $1.25^{-1} = 0.8$; $1.25^{-2} = 0.64$; $1.25^{-3} = 0.512$; $1.25^{-4} = 0.4096$; $1.25^{-5} = 0.32768$; $1.25^{-6} = 0.262144$; $1.25^{-7} = 0.2097152$; and so on. Therefore a $u_{ij} = \frac{1}{3}$ would be rounded up to 0.4096; a $u_{ij} = \frac{1}{4}$ to 0.262144, and so on.

We note that the task system obtained as a result of the transformation depends upon the task system that is to be transformed, and the value of the constant ϵ, but *not* upon the characteristics of the particular platform on which it is to be partitioned. In the remainder of this paper, we will attempt to partition the task system obtained by transforming the original task system; the following lemma asserts that doing so costs us a processor speedup no greater than $(1+\epsilon)$:

Lemma 1 *If there is no feasible partitioning of the transformed task system upon a given platform, then there is no feasible partitioning of the original system upon a platform in which each processor is* $\left(\frac{1}{1+\epsilon}\right)$ *times as fast.* ■

Let E denote the number of integer powers of $(1+\epsilon)$ in $(\epsilon, 1]$. For our example of $\epsilon = 0.25$, $E = 7$. (Since $(1+\epsilon)^{-7}$, at 0.2097152, is $\leq \epsilon$, the integer powers of 1.25 in $(0.25, 1]$ are $0, 1, 2, 3, 4, 5$, and 6.)

Observing more generally that $(1+\epsilon)^0 = 1$ and

$$\left((1+\epsilon)^{-x} < \epsilon\right) \Leftrightarrow \left(x > \frac{\log(1/\epsilon)}{\log(1+\epsilon)}\right),$$

we conclude that

$$E = \left\lceil \frac{\log(1/\epsilon)}{\log(1+\epsilon)} \right\rceil. \quad (1)$$

6 Characterizing feasible partitionings

In this section we consider *any* task system in which all task utilization values are integer powers of $(1+\epsilon)$, that can be feasibly partitioned on the given multiprocessor platform. Let us consider feasible partitionings of such task systems on the platform. We first classify tasks as big tasks or small tasks, as follows

Definition 2 (Big and small tasks) *We say that the i'th task is* big *for processor j if $u_{ij} > \epsilon$; otherwise, it is* small *for processor j.* ■

We seek to determine the number of different ways (or "patterns") in which big tasks can be arranged upon the processors in such a feasible partitioning. We will show (Lemma 2) that this number is bounded from above by a polynomial in the number of processors m.

6.1 A single processor

First, let us consider any one processor on the platform in this feasible partitioning. For each $t, 0 \leq t < E$, let π_t denote the number of big tasks with utilization equal to $(1+\epsilon)^{-t}$ that are assigned to this processor in this feasible partitioning. We refer to the vector $\langle \pi_0, \pi_1, \ldots, \pi_{E-1} \rangle$ as the ***pattern of big tasks*** on this processor. We say this pattern induces π_t ***slots*** of size $(1+\epsilon)^{-t}$ on the processor for each $t,\ 1 \leq t < E$. Each such slot accommodates one task with utilization $(1+\epsilon)^{-t}$.

For our running example of $\epsilon = 0.25$, suppose that a feasible partitioning were to assign two big tasks with utilization $1.25^{-6} = 0.262144$ each, and one big task with utilization $1.25^{-4} = 0.4096$, on a processor. This information could be represented by the pattern of big tasks $\langle 0, 0, 0, 0, 1, 0, 2 \rangle$.

In addition, this pattern π leaves an ***idle capacity*** equal to a fraction

$$1 - \sum_{t=0}^{E-1} \frac{\pi_t}{(1+\epsilon)^t} \tag{2}$$

of the capacity of the processor; this idle capacity may be used for accommodating tasks that are small tasks on that processor.

For the example of $\epsilon = 0.25$ and the pattern of big tasks $\langle 0, 0, 0, 0, 1, 0, 2 \rangle$; this idle capacity equals

$$1 - (1 \times 0.4096 + 2 \times 0.262144) \quad = \quad 0.066112.$$

Notation. Let Q denote the total number of possible patterns of big tasks on a processor, that could occur in any feasible partitioning. Since each big task on a processor has utilization $> \epsilon$ on that processor, there can be no more than $\lfloor \frac{1}{\epsilon} \rfloor$ big tasks on a processor; therefore, none of the π_t values in any pattern of big tasks may take on a value larger than $\lfloor \frac{1}{\epsilon} \rfloor$. It therefore follows that

$$Q \leq \big(1 + \left\lfloor \frac{1}{\epsilon} \right\rfloor\big)^E. \tag{3}$$

Note that this upper bound on the number of possible different patterns of big tasks on a processor is a *constant* for a given value of the constant ϵ. Let us define some arbitrary ordering on these Q possible patterns, so that it makes sense to speak of the ℓ'th pattern of big tasks on a processor for each $\ell, 1 \leq \ell \leq Q$.

1. A pattern of big tasks on a *single* processor: $\langle \pi_0, \ldots, \pi_{E-1} \rangle$
   - There are at most Q such possible distinct patterns, where Q is bounded as in Equation 3
2. Patterns of big tasks on *all the processors of a particular type*: $\langle d_1, d_2, \ldots, d_Q \rangle$
   - There are at most m^Q such possible distinct patterns
3. Patterns of big tasks on *all the processors* of the platform
   - There are at most m^{kQ} such possible distinct patterns (Equation 4)

Table 2: Summarizing patterns of big tasks

6.2 Processors of a single type

Next, consider all the processors on the platform of any one *type* in the feasible partitioning. We can represent the combinations of patterns of big tasks on these processors as a vector $\langle d_1, d_2, \ldots, d_Q \rangle$ where each d_ℓ is a non-negative integer. This vector denotes that there are d_ℓ processors of this type, upon which the task-assignment corresponds to the ℓ'th pattern of big tasks on a processor, for each $\ell, 1 \leq \ell \leq Q$.

Since each d_ℓ is bounded from above by the total number of processors of this type, which is in turn bounded from above by the total number of processors m, there are at most m^Q possible such vectors that could occur in any feasible partitioning.

6.3 All the processors

Finally, let us consider all the processors on the platform in the feasible partitioning. Since there are κ distinct types of processors, the total number of patterns of big tasks on the platform that could occur in any feasible partitioning is bounded from above by

$$m^{\kappa Q}. \tag{4}$$

The various kinds of patterns defined above are summarized in Table 2.

Observe that, since Q is constant for constant ϵ, the possible number of distinct patterns of big tasks on all the processors of the platform is polynomial in m for a given value of the constant ϵ. This is formalized in the following lemma:

Lemma 2 *For constant ϵ, there are at most polynomially many possible distinct patterns of big tasks on all the processors of the platform.* ■

7 Obtaining an (approximate) feasible partitioning

Upon a given multiprocessor platform, Lemma 2 above shows that there are at most polynomially many possible distinct patterns of big tasks on all the processors that could correspond to feasible partitionings. We now seek to determine whether one of these could correspond to a feasible partitioning of the task system that we wish to partition. However we do not know which of these patterns (if any) may be the one corresponding to a feasible partitioning of our task system; therefore, we will consider each in turn. In considering a pattern we construct, in polynomial time, an ILP such that a solution to this ILP exists if and only if the pattern corresponds to a feasible partitioning of our particular task system upon a platform in which each processor is slower[2] by a factor $(1-\epsilon)$. The manner in which this ILP is constructed is described in Section 7.1. We then seek to solve a "relaxation" of the ILP, which may allocate a task across multiple processors (with the fractions of each task allocated across all the processors summing to one). If we obtain such a solution, we move on to the next step listed below; else, we consider the next pattern. If we exhaust all the patterns without finding a solution to the LP relaxation of the ILP corresponding to any of the patterns then we report failure. We will show that such failure implies that the task system cannot be partitioned in a feasible manner upon a platform in which each processor is slower by a factor $(1-\epsilon)$.

If we did not report failure above, we will have obtained a fractional solution to the LP relaxation of the ILP corresponding to some pattern of big tasks.

1. We first use this fractional solution to construct a bipartite graph, and to identify a fractional matching in this bipartite graph – this step is detailed in Section 7.2.
2. Next, we use the algorithm of Birkhoff [3] to obtain an integral matching in the graph, also in polynomial time.
3. Finally we use this integer matching to deduce a partitioning of our particular task system – Section 7.3 explains how this is done.

7.1 The integer linear program

We first introduce some *notation:*

- Let S denote all the slots induced on all the processors by the particular pattern of big tasks under consideration. We note that since each processor may have no more than $\lfloor \frac{1}{\epsilon} \rfloor$ slots, there are no more than $m \lfloor \frac{1}{\epsilon} \rfloor$ slots across all the processors; this is polynomial in m.
- Let IDLE_j denote the fraction of the capacity of the j'th processor that is not assigned to big tasks by the combination of patterns under consideration; the value of IDLE_j is determined according to Expression 2.

The linear program is on the following **variables**:

- A variable z_{is} for each task i and each slot $s \in S$ such that the i'th task would fit exactly into slot s. That is, if slot s is on the j'th processor and is of size

[2]We will see that we need to consider slower processors here than the ones actually available, since some error is introduced in obtaining a feasible partitioning from this ILP in polynomial time. This is only to be expected since solving ILPs is known to be NP-hard.

$(1+\epsilon)^{-t}$, then we define the variable z_{is} only for those $i, 1 \leq i \leq n$, for which $u_{ij} = (1+\epsilon)^{-t}$.

Informally speaking, the variable z_{is} is to take on the value one if the i'th task is assigned to the s'th slot, and zero otherwise.

- A variable x_{ij} for each task i and each processor j, for which $u_{ij} \leq \epsilon$. (That is, task i is a small task on the j'th processor.)

  Informally speaking, the variable x_{ij} is to take on the value one if the i'th task is assigned to the idle capacity on the j'th processor, and zero otherwise.

Note that there are no more than $n \times (m/\epsilon) + n \times m$ variables, which, for constant ϵ, is polynomial in n and m.

The **constraints** in the LP are as follows:

$$\sum_{j=1}^{m} x_{ij} + \sum_{s \in S} z_{is} = 1 \qquad \text{for each } i, 1 \leq i \leq n \tag{5}$$

$$\sum_{i=1}^{n} z_{is} \leq 1 \qquad \text{for each } s \in S \tag{6}$$

$$\sum_{i=1}^{n} x_{ij} u_{ij} \leq \text{IDLE}_j - \epsilon \qquad \text{for each } j, 1 \leq j \leq m \tag{7}$$

$$x_{ij} \geq 0 \qquad \text{for each } i, j$$

$$z_{is} \geq 0 \qquad \text{for each } i, s$$

Constraint 5 asserts that each task be assigned; Constraint 6, that each slot gets no more than one task. Constraint 7 checks that all the small tasks assigned to a processor are able to fit into the amount of capacity left over on that processor by the big tasks. *Note the "$-\epsilon$" term on the RHS of Constraint 7*: this reflects the assumption that the ILP represents a solution on processors that are slower by a factor $(1-\epsilon)$, and hence a fraction ϵ of the capacity of each processor remains unused in this feasible partitioning.

It is evident that each integral solution to the ILP with Constraints 5-7 corresponds to a feasible partitioning of the task system upon a platform in which each processor is slower by a factor $(1-\epsilon)$; this is formalized in the following lemma:

Lemma 3 *The ILP represented by Expressions 5-7 has integral solutions for all the x_{ij} and all the z_{is} variables if and only if the task system (obtained after the transformation of Section 5) has a feasible partitioning according to the enumerated pattern upon a platform in which each processor is slower by a factor* $(1-\epsilon)$. ■

Unfortunately, solving an ILP is highly intractable — it is NP-hard in the strong sense. However if the restriction that all the variables take on integer values is removed, then a solution to the relaxed ILP (which is now just a linear program) can be obtained in polynomial time – see Section 3.1. We therefore obtain such a fractional solution to the LP; *let $\hat{x}_{ij}$ ($\hat{z}_{is}$, resp.) denote the value assigned to the variable x_{ij} (z_{is}, resp.) in this non-integral solution to the relaxed ILP.*

7.2 The graph, and the fractional matching

Given the fractional solution to the LP denoted by the $\hat{x}_{ij}$'s and the $\hat{z}_{is}$'s as described above, our next step is to concurrently (i) build a bipartite graph, and (ii) identify a fractional matching in this bipartite graph.

We first describe the procedure by which we determine the ***nodes*** of this graph.

- *task nodes.* We will have a single task node v_i for each task i, $1 \leq i \leq n$.
- *slot nodes.* We will have a single slot node w_s for each slot $s \in S$.
- *proc nodes.* We will have several proc nodes for each processor j. Let k_j be defined as follows:
$$k_j := \left\lceil \sum_i \hat{x}_{ij} \right\rceil$$
That is, k_j denotes the ceiling of the sum of the fractional assignments of tasks that are small on processor j, that get made to processor j in the fractional solution. For each processor j we will have k_j proc nodes, denoted $u_j^{(1)}, u_j^{(2)}, \ldots, u_j^{(k_j)}$.

Intuitively, each task node corresponds to a task, and each slot node and each proc node corresponds to a "place" where a task can be accommodated. (Since the solution to the LP had assigned a total of $\sum_i \hat{x}_{ij}$ tasks on the idle capacity on the j'th processor and we seek an integral assignment, this idle slot is assigned "places" capable of accommodating $k_j = \lceil \sum_i \hat{x}_{ij} \rceil$ tasks.) An edge between a task node and a slot node or a proc node denotes that the task is assigned to that "place;" a fractional weight assigned to that edge denotes that a fraction of the task is assigned to that "place."

We now describe how we determine the ***edges*** of the graph. We will see that all edges connect task nodes (v_i's) to slot nodes or proc nodes (w_s's or $u_j^{(k)}$'s) – the graph is thus indeed a bipartite graph. We also describe how we assign ***weights*** to these edges such that these weights comprise a fractional matching for the set of task nodes $\{v_i\}_{i=1}^n$. That is, the weights are assigned such that the sum of the weights of edges incident on each task node exactly equals one, and the sum of the weights of edges incident on each slot node and each proc node is no larger than one.

Edges corresponding to large tasks. For each $\hat{z}_{is} > 0$ (i.e., for each variable z_{is} that was assigned a value > 0 in the fractional solution), we add the edge (v_i, w_s) and assign it a weight equal to $\hat{z}_{is}$. (Informally, we are assigning a fraction $\hat{z}_{is}$ of the i'th task to the slot s.)

Since the $\hat{x}_{ij}$ and $\hat{z}_{is}$ values comprise a solution to the LP, Condition 6 ensures that no node w_s will have incident edges with weights summing to more than one.

Edges corresponding to small tasks. The procedure here is more elaborate. We describe it below, followed by an illustrative example (Example 7.2); the reader may find it convenient to follow the example concurrently with the description.

Let us consider a particular processor j.

```
1   $i \leftarrow 1$; $k' \leftarrow 1$; $need \leftarrow \hat{x}_{ij}$; $avail \leftarrow 1$
    $\triangleright$ $need$ denotes the remaining edge-weight that needs to be assigned to edges inci-
       dent on $v_i$. $avail$ denotes the remaining edge-weight that can be assigned to edges
       incident on $u_j^{(k')}$.
2   while $(i \leq n_j)$
3       do if $(need \leq avail)$
              then $\triangleright$ A single additional edge from $v_i$ to $u_j^{(k')}$ will suffice
4                  add edge $(v_i, u_j^{(k')})$ with weight $need$ (i.e., $\hat{x}_{ij}^{(k')} \leftarrow need$)
5                  $avail \leftarrow avail - need$
6                  $i \leftarrow i + 1$;$need \leftarrow \hat{x}_{i,j}$ $\triangleright$ Consider the next task
              else $\triangleright$ An edge from $v_i$ to $u_j^{(k')}$ is not enough; will need to add an edge
                     from $v_i$ to $u_j^{(k'+1)}$
7                  if $(avail > 0)$
                     then $\triangleright$ An edge from $v_i$ to $u_j^{(k')}$ for some of the $need$; rest on an
                            edge to $u_j^{(k'+1)}$
8                         add edge $(v_i, u_j^{(k')})$ with weight $avail$ (i.e., $\hat{x}_{ij}^{(k')} \leftarrow avail$)
9                         $need \leftarrow need - avail$
10                 $k' \leftarrow k' + 1$; $avail \leftarrow 1$ $\triangleright$ Go to the next proc node
```

Figure 1: Pseudo-code for adding edges corresponding to the assignment of small tasks

- If $\hat{x}_{ij} > 0$ then we will add edges from node v_i to one or two of the nodes in $\{u_j^{(1)}, u_j^{(2)}, \ldots, u_j^{(k_j)}\}$, in the manner detailed below. We will see that the weights assigned to these edges sum to $\hat{x}_{ij}$.

- Suppose that there are n_j such tasks (i.e., there are n_j different i such that $\hat{x}_{ij} > 0$).

- For notational convenience, let us (locally – i.e., for the purposes of this argument only) *rename* these n_j tasks so that they are indexed as $1, 2, \ldots, n_j$ in non-increasing order of utilization on the j'th processor; i.e., with the property that $u_{ij} \geq u_{(i+1)j}$ for all $i, 1 \leq i < n_j$.

- We will now add some edges between the v_i's and the $u_j^{(k)}$'s, and assign weights to these edges. Let $\hat{x}_{ij}^{(k)}$ denote the weight we assign to the edge between nodes v_i and $u_j^{(k)}$ (informally, a fraction $\hat{x}_{ij}^{(k)}$ of the i'th task will be assigned to the k'th "place" on the idle capacity left over on processor j by the slots S). Since $\hat{x}_{ij}$ denotes the fraction of the i'th task will be assigned to the idle capacity left over on processor j by the slots S, we must ensure that

$$\sum_{k=1}^{n_j} \hat{x}_{ij}^{(k)} = \hat{x}_{ij}$$

  Figure 1 describes the procedure by which these edges are added. Informally,

  - The tasks are considered in non-increasing order of their utilizations on the j'th processor (this follows from the renaming step as described above).
  - In "considering" a task i, the objective is to add edges from v_i to the nodes $\{u_j^{(k)}\}_{k=1}^{k_j}$ with weights summing to $\hat{x}_{ij}$. Furthermore, the sum of the weights of the edges incident on any node $u_j^{(k)}$ must not exceed 1.
  - During the process of adding edges such that these objectives are fulfilled, an invariant is maintained: *at most one of the proc nodes – the $u_j^{(k)}$'s – has incoming edges summing to more than zero but less than one*.
  - Let $u_j^{(k')}$ denote the node of the invariant, if any. Initially, $k' \leftarrow 1$.
    If an edge of weight $\hat{x}_{ij}$ can be added from the node v_i corresponding to the task under consideration to this node $u_j^{(k')}$ without causing the sum of the weights of the edges incident on $u_j^{(k')}$ to exceed 1, then this edge is added and the task $(i + 1)$ becomes the task under consideration. Otherwise, two edges are added:
    1. an edge of as much weight as can be added without causing the sum of the weights of the edges incident on $u_j^{(k')}$ to exceed 1, is added from node v_i to node $u_j^{(k')}$; and
    2. another edge of weight equal to $\hat{x}_{ij}$ minus the weight assigned to the edge in the step above, is added from node v_i to node $u_j^{(k'+1)}$.

Note that the invariant is maintained, since node $u_j^{(k')}$ now has incoming edges summing to one while node $u_j^{(k'+1)}$ may have incoming edges summing to more than zero but less than one.

Example Let us focus upon a particular j. For this particular j, suppose that the LP solution has resulted in three of the $\hat{x}_{ij}$ values being $0.6, 0.7$, and 0.75, with the remaining values all zero.

- In accordance with the "renaming" mentioned above, we will index these three tasks by 3, 2, and 1 respectively, so that $\hat{x}_{1j} = 0.75$, $\hat{x}_{2j} = 0.7$, and $\hat{x}_{3j} = 0.6$.
- Since $\lceil 0.6 + 0.7 + 0.75 \rceil = 3$, there will be three nodes $u_j^{(1)}$, $u_j^{(2)}$, and $u_j^{(3)}$.
- Since $\hat{x}_{1j} = 0.75$, we assign an edge of weight 0.75 between v_1 and $u_j^{(1)}$. That is, $\hat{x}_{1j}^{(1)} = 0.75$.
- Next, we consider $\hat{x}_{2j}$, which equals 0.7. We therefore assign an edge of weight $(1 - 0.75) = 0.25$ between v_2 and $u_j^{(1)}$, and an edge of weight $(0.7 - 0.25) = 0.45$ between v_2 and $u_j^{(2)}$. That is, $\hat{x}_{2j}^{(1)} = 0.25$ and $\hat{x}_{2j}^{(2)} = 0.45$.
- Finally, we consider $\hat{x}_{3j}$, which equals 0.6. We therefore assign an edge of weight $(1 - 0.45) = 0.55$ between v_3 and $u_j^{(2)}$, and an edge of weight $(0.6 - 0.55) = 0.05$ between v_3 and $u_j^{(3)}$. That is, $\hat{x}_{3j}^{(2)} = 0.55$ and $\hat{x}_{3j}^{(3)} = 0.05$.

■

7.3 Determining the task assignment

From the fractional matching defined above, we use the algorithm of [3] to determine, in polynomial time, an *integral* matching. This matching satisfies the property that each task-node v_i is matched to either a single slot node w_s for some $s \in S$, or a single proc node $u_j^{(k)}$ for some j, k with $1 \le j \le m$ and $1 \le k \le k_j$.

From the integral matching thus obtained, we determine the assignment of tasks to processors according to the following two rules.

Rule 1: If task node v_i is matched to a slot node w_s, then the i'th task is assigned to the processor on which the slot s was induced.

Since there are only edges from task nodes to slot nodes on which the task would fit, it is evident that

- all tasks assigned in this manner fit on the processor to which they are assigned; and
- these assignments leave a fraction IDLE_j of the capacity of the j'th processor available for assigning to small tasks.

Rule 2: If task node v_i is matched to a proc node $u_j^{(k)}$, then the i'th task is assigned to the j'th processor.

We will now show that for all j, $1 \leq j \leq m$, the total utilization of all the small tasks assigned to the j'th processor in this manner does not exceed IDLE_j.

Notation: Recall that $\hat{x}_{ij}^{(k)}$ denotes the weight that was assigned to the edge between task node v_i and proc node $u_j^{(k)}$, in the fractional matching that was obtained in Section 7.2 above. Let $\tilde{x}_{ij}^{(k)}$ denote the weight assigned to the edge between task node v_i and proc node $u_j^{(k)}$ in the integer matching obtained by applying the algorithm of [3] to the fractional matching; it is evident that each $\tilde{x}_{ij}^{(k)}$ is either zero or one. Let $\tilde{x}_{ij}$ denote $\sum_{k=1}^{k_j} \tilde{x}_{ij}^{(k)}$; i.e., $\tilde{x}_{ij}$ is the sum of the weights on the edges from v_i to all of $\{u_j^{(1)}, u_j^{(2)}, \ldots, u_j^{k(j)}\}$ in the integer matching.

We note that for a given i and j, the fractional matching that was obtained in Section 7.2 may have edges from v_i to two "consecutive" proc nodes $u_j^{(k)}$ and $u_j^{(k+1)}$. The integral matching will match v_i to at most one of these two proc nodes.

- Recall that in adding edges between task nodes and the proc nodes corresponding to a particular processor j, we first ordered the task nodes in non-increasing order of their utilizations on the j'th processor. Therefore if task node $v_{i'}$ is fractionally matched to proc node $u_j^{(k-1)}$ and task node v_i is fractionally matched to proc node $u_j^{(k)}$ in the fractional matching, then $u_{i'j} \geq u_{ij}$.

- Therefore if task node v_i is matched with proc node $u_j^{(k)}$ in the *integral* matching (i.e. $\tilde{x}_{ij}^{(k)} = 1$), it must be the case that u_{ij} is at most the utilizations of all tasks that are matched with proc node $u_j^{(k-1)}$ in the fractional matching. That is, for all $k \geq 2$
$$\sum_{i=1}^{n} \tilde{x}_{ij}^{(k)} u_{ij} \leq \sum_{i=1}^{n} \hat{x}_{ij}^{(k-1)} u_{ij} \tag{8}$$

Now the cumulative utilization of all the small tasks on the j'th processor is given by

$$\begin{aligned}
\sum_{i=1}^{n} \tilde{x}_{ij} u_{ij} &= \sum_{i=1}^{n}\sum_{k=1}^{k_j} \tilde{x}_{ij}^{(k)} u_{ij} \text{ (Since } \tilde{x}_{ij} = \textstyle\sum_{k=1}^{k_j} \tilde{x}_{ij}^{(k)}\text{)} \qquad (9)\\
&= \sum_{k=1}^{k_j}\sum_{i=1}^{n} \tilde{x}_{ij}^{(k)} u_{ij}\\
&\leq \max\{u_{ij} \mid \hat{x}_{ij} > 0\} + \sum_{k=2}^{k_j}\sum_{i=1}^{n} \tilde{x}_{ij}^{(k)} u_{ij}\\
&\leq \epsilon + \sum_{k=2}^{k_j}\sum_{i=1}^{n} \tilde{x}_{ij}^{(k)} u_{ij} \text{ (Since task } i \text{ is small for processor } j\text{)}\\
&\leq \epsilon + \sum_{k=1}^{k_j-1}\sum_{i=1}^{n} \hat{x}_{ij}^{(k)} u_{ij} \text{ (By Equation 8)}\\
&\leq \epsilon + \sum_{i=1}^{n} \hat{x}_{ij} u_{ij}\\
&\leq \epsilon + (\text{IDLE}_j - \epsilon) \text{ (Since the } \hat{x}_{ij}\text{'s satisfy Constraint 7 of the LP)}\\
&\leq \text{IDLE}_j
\end{aligned}$$

We have thus shown that the total utilization of all the small tasks assigned to the j'th processor can be accommodated in the idle capacity IDLE_j left over by the big tasks. Lemma 4 follows.

Lemma 4 *If the LP-relaxation of the ILP of Section 7.1 has a solution, then our task assignment strategy (as described by Rules 1 and 2 above) obtains a feasible partitioning.* ■

8 Summary of the algorithm

Given an implicit-deadline sporadic task system to be partitioned among the processors of a limited unrelated multiprocessor platform,

1. We round up all the utilization parameters, to obtain a transformed task system.
2. We (implicitly or explicitly) enumerate all the patterns of big tasks upon the given platform, that correspond to feasible partitionings of task systems. We consider each separately; in considering a pattern,
   (a) We set up an ILP representing a feasible partitioning of the transformed task system upon a platform in which each processor is slowed down by a factor $(1 - \epsilon)$.
   (b) We solve the LP-relaxation of this ILP. If successful, we proceed to the next step. If the LP-relaxation has no solution, we consider the next pattern

of big tasks; if we have considered all the patterns of big tasks, we declare failure and return.

3. From the solution to the LP-relaxation of the ILP, we construct a bipartite graph and define a fractional matching on this graph.

4. We apply the algorithm of [3] (mentioned in Theorem 1 above) to determine an integer matching from the fractional one.

5. We use this integer matching to determine a feasible partitioning of the task system.

Our algorithm makes the following performance guarantee:

Theorem 2 *If our polynomial-time algorithm fails to obtain a feasible partitioning of a task system upon a given platform, then no algorithm can partition the task system on a platform in which each processor is slower by a factor* $\left(\frac{1-\epsilon}{1+\epsilon}\right)$.

Proof Suppose that our algorithm fails to find a feasible partitioning. By Lemma 4, the ILP constructed in Section 7.1 does not have a solution. Hence by Lemma 3, the (transformed) task system does not have a feasible partitioning on a platform on which each processor is slower by a factor $(1-\epsilon)$. By applying Lemma 1 we therefore conclude that the original task system cannot have a feasible partitioning on a platform on which each processor is slower by a factor $\left(\frac{1-\epsilon}{1+\epsilon}\right)$. ■

9 Context and Conclusions

As multicore architectures become ubiquitous, chip makers are trying to distinguish their products by providing specialized hardware components that accelerate particular computations. Examples include multicore CPUs with specialized graphics processing cores, specialized signal-processing cores, specialized floating-point units, customizable FPGAs, etc. These multicore CPUs offer a fresh challenge to real-time scheduling theory: task assignment on such multiprocessor platforms is known to be highly intractable. This fact has given rise to research into efficient (polynomial-time) algorithms for partitioning real-time workloads that are not exact, but nevertheless exhibit reasonably good performance.

Scheduling problems, partitioned or otherwise, are inherently decision problems: either a system is schedulable or not. However when answering the decision problem exactly is provably intractable, it is useful to define an optimization version in order to be able to quantitatively discuss the effectiveness of different approximate scheduling algorithms. For partitioned scheduling, one such optimization version – the one we use in this paper – asks: what is the minimum speed that the processors on the platform must have, in order for the system to be schedulable?

Analogously to the classification of problems as polynomial-time, NP-complete, PSPACE complete, etc., there is a classification of NP-hard optimization problems according to the difficulty of obtaining approximate solutions to these problems in polynomial-time. In particular, an NP-hard minimization problem is said to be in the class **APX** if there is a constant c such that some polynomial-time algorithm can obtain a solution to any problem instance that is no more than c times the cost of the (optimal) minimum-cost solution. APX problems for which there exist polynomial-time algorithms that can obtain a solution to any problem instance that is no more than c times the cost of the (optimal) minimum-cost solution for all $c > 1$ are said to be in the class **PTAS**. It is good to show that an NP-hard optimization problem is in the class APX, even better to show that it is in the class PTAS.

From the results in [10], it follows that the problem of partitioning implicit-deadline sporadic task systems on unrelated multiprocessor platforms is *not* in PTAS (assuming that P $\neq$ NP). However there is an interest in studying unrelated multiprocessor platforms with large core-counts, in which all the cores are of only a few distinct types — this interest was articulated in [1], motivated by currently-available and soon-to-be-available multicore CPUs. For such limited unrelated multiprocessor platforms, we have shown here that partitioning implicit-deadline sporadic task systems is indeed in PTAS. We reiterate that we do not claim great practical relevance to the PTAS we have derived; instead, it should be thought of primarily as an algorithmic core that gets adapted and fine-tuned prior to its use in any particular application domain, according to the characteristics of the task systems arising in those particular applications.

References

[1] ANDERSSON, B., RARAVI, G., AND BLETSAS, K. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In *Proceedings of the Real-Time Systems Symposium* (San Diego, CA, 2010), IEEE Computer Society Press, pp. 239–248.

[2] BARUAH, S. Partitioning real-time tasks among heterogeneous multiprocessors. In *Proceedings of the Thirty-third Annual International Conference on Parallel Processing* (Montreal, Canada, August 2004), IEEE Computer Society Press, pp. 467–474.

[3] BIRKHOFF, G. Tres observaciones sobre el algebra lineal. In *Revista Facultad de Ciencias Exactas, Puras y Aplicadas Universidad Nacional de Tucuman, Serie A (Matematicas y Fisica Teorica)*, vol. 5. 1946, pp. 147–151.

[4] CHATTOPADHYAY, B., AND BARUAH, S. A lookup-table driven approach to partitioned scheduling. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)* (Chicago, 2011), IEEE Computer Society Press.

[5] DANTZIG, G. B. *Linear Programming and Extensions*. Princeton University Press, 1963.

[6] HOCHBAUM, D., AND SHMOYS, D. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM 34*, 1 (Jan. 1987), 144–162.

[7] HOCHBAUM, D., AND SHMOYS, D. A polynomial time approximation scheme for scheduling on uniform processors using the dual approximation approach. *SIAM Journal on Computing 17*, 3 (June 1988), 539–551.

[8] KARMAKAR, N. A new polynomial-time algorithm for linear programming. *Combinatorica 4* (1984), 373–395.

[9] KHACHIYAN, L. A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR 244* (1979), 1093–1096.

[10] LENSTRA, J.-K., SHMOYS, D., AND TARDOS, E. Approximation algorithms for scheduling unrelated parallel machines. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science* (Los Angeles, CA, Oct. 1987), A. K. Chandra, Ed., IEEE Computer Society Press, pp. 217–224.

[11] LIU, C., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[12] LOPEZ, J. M., DIAZ, J. L., AND GARCIA, D. F. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems: The International Journal of Time-Critical Computing 28*, 1 (2004), 39–68.

[13] PAPADIMITRIOU, C. H. On the complexity of integer programming. *Journal of the ACM 28*, 4 (1981), 765–768.

[14] WILLIAMSON, D., AND SHMOYS, D. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.