A lookup-table driven approach to partitioned scheduling*

Bipasa Chattopadhyay  
Sanjoy Baruah  
The University of North Carolina at Chapel Hill

Abstract

The partitioned preemptive EDF scheduling of implicit-deadline sporadic task systems on an identical multiprocessor platform is considered. Lookup tables, at any selected degree of accuracy, are pre-computed for the multiprocessor platform. By using these lookup tables, task partitioning can be performed in time polynomial in the representation of the task system being partitioned. Although the partitioning will not in general be optimal, the degree of deviation from optimality is bounded according to the degree of accuracy selected during the pre-computation of the lookup tables.

1 Introduction

Two different efficiency considerations play a role in determining scheduling strategies for embedded real-time systems. On the one hand, they should be efficient to implement; on the other, they should ensure efficient usage of the processor(s) that are being scheduled.

In this paper, we consider the partitioned preemptive EDF scheduling of implicit-deadline sporadic task systems (also known as Liu & Layland task systems [5]) on identical multiprocessor platforms. It is widely known (see, e.g., [6]) that such partitioning is equivalent to the bin-packing problem, and is hence highly intractable: NP-hard in the strong sense. Resource-allocation strategies that achieve optimal resource utilization are therefore likely to have very inefficient implementations.

In the search for resource allocation strategies that have efficient implementations, various heuristics for task partitioning have been studied and evaluated (see, e.g. [6]). The heuristics evaluated in such studies are those for which very efficient implementations are easily obtained (such as First-Fit, Best-Fit, Worst-Fit, First-Fit-Decreasing, etc.; please see [6] for a description of these heuristics in the context of task partitioning). These studies seek to determine sufficient schedulability conditions for these heuristics, and compare different heuristics by comparing their respective sufficient schedulability conditions. These results have proved very useful from the perspective of designing hard-real-time systems; however, they do not provide much insight as to how far removed the resource utilization of these different heuristics are, from optimality.

In other related work, Hochbaum and Shmoys [3] have designed a polynomial-time approximation scheme (PTAS) for the partitioning of implicit-deadline sporadic task systems that behaves as follows. Given any positive constant $\phi$, if an optimal algorithm can partition a given task system $\tau$ upon $m$ processors each of speed $s$, then the algorithm in [3] will, in time polynomial in the representation of $\tau$, partition $\tau$ upon $m$ processors each of speed $(1 + \phi)s$. This can be thought of as a resource augmentation result [4]: the algorithm of [3] can partition, in polynomial time, any task system that can be partitioned upon a given platform by an optimal algorithm, provided it (the algorithm of [3]) is given augmented resources (in terms of faster processors) as compared to the resources available to the optimal algorithm.

This is theoretically an immensely significant result since it allows us to perform task partitioning in polynomial time, to any (constant) desired degree of accuracy. However, the practical significance of this result is severely limited by the fact that the algorithm of [3] has very poor implementation efficiency in practice: the constants in the runtime expression for this algorithm are prohibitively large.

*Supported in part by NSF grants CNS 1016954, CNS 0834270, and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA9550-09-1-0549.
**This research.** We seek to apply the ideas in [3] to come up with an implementation that is efficient enough to be usable in practice. In brief, our approach is to split the computation needed to implement the algorithm of [3] into two parts: (i) a computation-intensive part that is done during the process of assembling the platform upon which the task system is to be implemented; and (ii) a far more efficiently-implementable part that is done when attempting to schedule any given sporadic task system upon the platform. The computation-intensive part is done only once when the multiprocessor platform is being synthesized, and the results stored in a lookup table. We envision that this table will be supplied along with the multiprocessor platform (in much the same manner that complex mathematical functions are sometimes implemented in lookup tables on modern processors). This table is therefore available for the designers of real-time systems when they seek to determine whether particular task systems can be scheduled under partitioned EDF upon this platform or not. Using this table, such a question can be answered very efficiently, in time that is a low-order polynomial in the number of tasks in the system.

**Organization.** The remainder of this paper is organized as follows. In Section 2, we formally describe the task and platform model that we will be assuming in the remainder of this paper. In Section 3, we provide a high-level overview of our proposed partitioning algorithm; a detailed description follows in Sections 4 and 5, with Section 4 detailing the manner in which the lookup tables are constructed and Section 5 describing how these tables are used to do task assignment. A running example is used throughout these two sections, helping to illustrate the workings of our proposed algorithm.

**2 Task Model**

As stated in Section 1, we are considering the partitioned preemptive EDF scheduling of implicit-deadline sporadic task systems on identical multiprocessor platforms. We now describe each of these terms in detail.

An implicit-deadline sporadic task [5], also sometimes referred to as a Liu & Layland task, is characterized by an ordered pair of parameters: a worst-case execution time (WCET) and a minimum inter-arrival separation (that is, for historical reasons, also called the period of the task). Let \( \tau_i \) denote an implicit-deadline sporadic task with WCET \( C_i \) and period \( T_i \). Such a task generates a potentially infinite sequence of jobs, with the first job arriving at any time and subsequent job-arrivals at least \( T_i \) time units apart. Each job has an execution requirement no greater than \( C_i \); this must be met by a deadline that occurs \( T_i \) time units after the job’s arrival.

We use the term utilization to denote the ratio of the WCET parameter of a task to its period: the utilization of \( \tau_i \), often denoted as \( u_i \), is equal to \( C_i/T_i \).

An implicit-deadline sporadic task system is comprised of a finite collection of sporadic tasks, each specified by the two parameters as described above.

An identical multiprocessor platform is comprised of a number \( m \) of processors \( (m > 1) \), each of which has the same computing capabilities as all the other processors in the platform. Our interest is in scheduling implicit-deadline sporadic task systems upon such an \( m \)-processor identical multiprocessor platform. The approach we will adopt is partitioned scheduling: the tasks in task system \( \tau \) are partitioned into \( m \) disjoint subsystems, with each subsystem being assigned to execute upon a distinct processor. We consider that the tasks thus assigned to each processor are scheduled on that processor using the preemptive Earliest Deadline First (EDF) scheduling algorithm [5, 2]. It follows from the results in [5] that a necessary and sufficient condition for the tasks assigned to each processor to be schedulable by EDF is that their utilizations sum to no more than the speed of the processor (assumed here to be equal to one).

**3 Overview of proposed approach**

The main idea behind our approach is to construct a lookup table (LUT) for each identical multiprocessor platform upon which we may intend to execute implicit-deadline sporadic task systems under partitioned EDF, and to supply this table along with the platform. Whenever a task system is to be partitioned upon this platform, this table may be used to determine the assignment of the tasks to the processors.

Now, the lookup table for a particular platform is constructed without knowledge of the task systems that will later need to be partitioned upon that platform. We do not therefore know, during table construction time, the exact characteristics of the tasks that will need to be assigned to the processors of the platform. Instead, the table is constructed assuming that the utilizations of all the tasks have values from within a fixed set of distinct values \( V \). When this lookup table is later used to actually perform partitioning of a given task system \( \tau \), each task in \( \tau \) may need to have its WCET parameter inflated so that the resulting task
utilization is indeed one of these distinct values in $V$. (The *sustainability* [1] property of preemptive uniprocessor EDF ensures that if the tasks with the inflated WCET’s are successfully scheduled, then so are the original tasks.) The challenge is to choose the distinct utilization values in $V$ in a clever manner, so that the amount of such inflation that is needed is bounded.

We will see that the larger the number of distinct utilization values we are permitted to have in the set $V$, the smaller the amount of inflation that is needed. Hence, an important design decision must be made prior to table-construction time: *How large a table will we construct?* This is expressed in terms of choosing a value for a parameter $\epsilon$ to the procedure that constructs the lookup table. Informally speaking, the smaller the value of $\epsilon$, the smaller the degree of rounding up that is needed, and the closer to optimal our subsequent task-assignment procedure will be. However, the size of the lookup table, and the time required to compute it, also depend on the value of $\epsilon$: the smaller the value, the larger the table-size (and the amount of time needed to compute it).

4 Constructing the lookup table

We now describe the construction of the lookup table that is to be provided with the multiprocessor platform. Recall that this table is constructed only once, at the time that the platform is being put together.

Let us suppose that we are given a multiprocessor platform consisting of $m$ unit-speed processors. The steps involved in constructing the lookup table for this platform are

1. Choosing a value for the parameter $\epsilon$, which determines the degree of accuracy.

2. Based on the value chosen for $\epsilon$, determining the utilization values that are to be included in the set $V$. (To make explicit the dependence of this set of utilization values upon the value chosen for $\epsilon$, we will henceforth denote this set as $V(\epsilon)$). Recall that during the process of actually partitioning implicit-deadline sporadic task systems upon this platform, we will be rounding up the actual utilizations of tasks to become equal to one of the utilization values in $V(\epsilon)$.

3. Determining the combinations of tasks with utilizations in $V(\epsilon)$ that can be scheduled together on a single processor.

4. Using these single-processor combinations to determine the combinations of tasks with utilizations in $V(\epsilon)$ that can be scheduled on $m$ processors.

Each of these steps is discussed in greater detail below, in Sections 4.1-4.4.

Throughout this section, we will illustrate the construction of the lookup table by means of an example. Let us suppose that the platform in this running example consists of 4 unit-speed processors (i.e., $m = 4$).

4.1 Choosing $\epsilon$

As stated in Section 3 above, the procedure for computing the lookup table must be provided a parameter $\epsilon$, which is a positive real number. A design decision must now be made, in the form of choosing a value for $\epsilon$. We will see later (Theorem 2) that the performance guarantee that is made by our partitioning algorithm is as follows: any task system that can be partitioned upon $m$ unit-speed processors by an optimal partitioning algorithm will be partitioned by our algorithm on $m$ processors each of speed $(1 + \epsilon)$. Hence, in choosing a value for $\epsilon$ we are in effect reducing the guaranteed utilization bound of each processor to equal $1/(1 + \epsilon)$ times the actual utilization; so the decision in choosing a value for $\epsilon$ essentially becomes: what fraction of the processor capacity are we willing to sacrifice? For instance, if we were willing to tolerate a loss of up to 10% of the processor utilization, $\epsilon$ would need to satisfy the condition

$$\frac{1}{1 + \epsilon} \geq 0.9$$

$$\Leftrightarrow \epsilon \leq \frac{1}{0.9} - 1$$

$$\Leftrightarrow \epsilon \leq \frac{1}{9}$$

As stated in Section 3, the size of the lookup table, and the time required to compute it, also depend on the value of $\epsilon$: the smaller the value, the larger the table-size (and the amount of time needed to compute it). Hence, $\epsilon$ is assigned the largest value consistent with the desired overall system utilization; in the example above, $\epsilon$ would in fact be assigned the value $1/9$.

Example 1 For our running example, let us choose the value 0.3 for the parameter $\epsilon$. (For an actual platform we

\[\text{We note that this “sacrifice” is only in terms of worst-case guarantees, as formalized in Theorem 2. It is quite possible that some of this sacrificed capacity can in fact be used during the partitioning of particular task systems.}\]
would typically choose a far smaller value, but this larger value is more useful for purposes of illustration here: for small values of \( \epsilon \), the sizes of the intermediate data structures are too large to be illustrative from a pedantic perspective.) With \( \epsilon \leftarrow 0.3 \), we are guaranteeing to achieve the same performance as an optimal algorithm would, on a platform consisting of the same number of processors each of speed or computing capacity equal to \( 1/(1+0.3) \), or \( \approx 0.77 \), of the speeds of the processors available to our algorithm; i.e., we are “sacrificing,” in the worst case, a bit less than a quarter of the platform’s computing capacity. (However, recall the point made in footnote 2, concerning the pessimism in this worst-case bound on the fraction of capacity that is sacrificed. This point is illustrated for our running example in Example 5, where the lookup table that we will construct can be used to successfully partition a task system with utilization \( \approx 3.6 \), in excess of the upper bound of \( 4 \cdot (1/1.3) \approx 3.077 \).)

4.2 Determining the utilization values

Once we have settled on a value for \( \epsilon \), we will use this value to determine which utilization values to include in the set \( V(\epsilon) \) of distinct utilization values that will be represented in the lookup table we construct. In choosing the members of \( V(\epsilon) \), the objective is to minimize the amount by which the utilizations of the tasks to be partitioned must be inflated, in order to become equal to one of the values in \( V(\epsilon) \).

The choice we make is to have \( V(\epsilon) \) be equal to the set of all real numbers of the form \( \epsilon \cdot (1 + \epsilon)^j \), for all non-negative integers \( k \) (up to the upper limit of one). Why are these particular values chosen? Recall that when the table is used to perform task partitioning, the actual task utilizations (which may take on any value) will be rounded up to the nearest value present in the set \( V(\epsilon) \). Suppose that an actual utilization \( u_i \) is just a bit greater than one of the values present in \( V \), say, \( \epsilon(1 + \epsilon)^j \) — this is depicted in the figure below by an “x”.

This utilization will be rounded up to \( \epsilon(1 + \epsilon)^{j+1} \); the fraction by which this utilization has been inflated is therefore

\[
\frac{\epsilon(1 + \epsilon)^{j+1}}{u_i} < \frac{\epsilon(1 + \epsilon)^{j+1}}{\epsilon(1 + \epsilon)^{j}} = (1 + \epsilon).
\]

Thus if each task’s utilization were to be inflated by this maximal factor, it follows that any collection of tasks with total utilization \( \leq 1/(1 + \epsilon) \) would have inflated utilization \( \leq 1 \), and would hence be determined, based on our lookup table, to fit on a single processor\(^3\).

Let us now determine \( |V(\epsilon)| \), the number of elements in the set \( V(\epsilon) \). We wish to include each positive real number \( \leq 1 \) that is of the form \( \epsilon(1 + \epsilon)^j \) for non-negative \( j \). Since

\[
\epsilon(1 + \epsilon)^j \leq 1
\]

\[
(1 + \epsilon)^j \leq (1/\epsilon)
\]

\[
\Rightarrow j \log(1 + \epsilon) \leq \log(1/\epsilon)
\]

\[
\Rightarrow j \leq \frac{\log(1/\epsilon)}{\log(1 + \epsilon)},
\]

we conclude that

\[
|V(\epsilon)| = \left\lfloor \frac{\log(1/\epsilon)}{\log(1 + \epsilon)} \right\rfloor + 1 \tag{1}
\]

Example 2 For our example (\( \epsilon = 0.3 \)), it may be verified that \( \frac{\log(1/0.3)}{\log(1+0.3)} \approx 4.589 \). Hence by Equation 1 there are \( \left\lfloor 4.589 \right\rfloor + 1 = 5 \) elements in \( V(0.3) \). We therefore have 5 distinct utilization values to consider: for \( j = 0, 1, 2, 3, \text{ and } 4 \). (The value of 1.114 for \( j = 5 \) is too large, as are the values for all \( j > 5 \).) These elements are computed as follows:

\[
\begin{array}{c|c}
 j & \text{util. value} (0.3 \cdot (1.3)^j) \\
 0 & 0.3000 \\
 1 & 0.3900 \\
 2 & 0.5070 \\
 3 & 0.6591 \\
 4 & 0.8568 \\
 5 & 1.114 \\
\end{array}
\]

Hence

\[
V(0.3) = \{ 0.3000, 0.3900, 0.5070, 0.6591, 0.8568 \}.
\]

4.3 Determining legal single-processor configurations

In Section 4.2 above, we have determined the distinct utilization values in the set \( V(\epsilon) \). We now seek to determine all

\(^3\)Note that this argument does not hold for actual utilizations — the \( u_i \) in the figure — less than \( \epsilon/(1 + \epsilon) \). If a task with utilization \( u_i \) arbitrarily close to zero \((u_i \to 0^+)\) were to have its utilization rounded up to \( \epsilon/(1 + \epsilon) \), the inflation factor would be \( (\epsilon/(1 + \epsilon)) + u_i \), which approaches \( \infty \) as \( u_i \to 0 \). We will see that the task-assignment procedure of Section 5 handles tasks with utilization \( \leq \epsilon/(1 + \epsilon) \) differently.
the different ways in which a single processor can be packed with tasks of these utilizations. We refer to these as single-processor configurations. For reasons of efficiency in storage (and subsequent lookup), we will seek only the maximal configurations of this kind: a single-processor configuration is said to be a maximal one if no additional task (also with utilization $\in V(\epsilon)$) can be added without the sum of the utilizations exceeding the capacity of the processor:

**Definition 1 (Single-proc. configuration)** For a given value of $\epsilon$, a single-processor configuration is a $|V(\epsilon)|$-tuple

$$\langle x_1, x_2, \ldots, x_{|V(\epsilon)|} \rangle$$

of non-negative integers, satisfying the constraint that

$$\left( \sum_{i=1}^{|V(\epsilon)|} (x_i \cdot (1 + \epsilon)^i - 1) \right) \leq 1. \quad (2)$$

The single-processor configuration $\langle x_1, x_2, \ldots, x_{|V(\epsilon)|} \rangle$ is maximal if

$$\left( \sum_{i=1}^{|V(\epsilon)|} (x_i \cdot (1 + \epsilon)^i - 1) \right) > (1 - \epsilon) \quad (3)$$

(thereby implying that no task can be added to this single-processor configuration without exceeding the processor’s capacity). ■

Our objective is to determine a list $L_1(\epsilon)$ of all possible maximal single-processor configurations for the selected value of $\epsilon$ (here, the subscript “1” denotes the number of processors; in section 4.4 below, we will describe how we construct the list $L_i(\epsilon)$ for $m$ processors).

Since there are only finitely many distinct utilization values in $V(\epsilon)$ (the exact number is as determined by Equation 1), all the elements of $L_1(\epsilon)$ can in principle be determined by exhaustive enumeration: simply try all $|V(\epsilon)|$-tuples with the $i$th component no larger than $(1/(\epsilon \cdot (1 + \epsilon)^i - 1))$, adding the ones that satisfy Inequalities 2 and 3 to $L_1(\epsilon)$. Such a procedure has run-time exponential in $(1/\epsilon)$; the smaller the value of $\epsilon$, the greater the run-time of the procedure (and of the length of $L_1(\epsilon)$ — the number of maximal single-processor configurations found). Although this can be quite high for small $\epsilon$, we point out that

- This run-time is incurred only once, when the multiprocessor platform is being put together. Once the list $L_1(\epsilon)$ has been constructed, it can be stored and repeatedly reused for doing task partitioning.

<table>
<thead>
<tr>
<th>Config. ID</th>
<th>0.3000</th>
<th>0.3900</th>
<th>0.5070</th>
<th>0.6591</th>
<th>0.8568</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. All the maximal single-processor configurations for the example.

- Although the size of $L_1(\epsilon)$ is indeed exponential in $(1/\epsilon)$, our experiments reveal that this size is quite reasonable in practice for values of $\epsilon$ that are not too small. We have computed these lists for various values of $\epsilon$: for instance, choosing $\epsilon = 1/9$ (which is equivalent to sacrificing at most 10% of each processor’s capacity) yields a list of approximately 3200 maximal single-processor configurations. Given current memory costs, look-up tables of sizes far larger than this are quite viable — consider the lookup tables used in, e.g., floating-point co-processors for speeding up the computation of operations such as $\sin$, $\cos$, $\log$, etc., which are often tens of megabytes large.

- Several simple and straightforward counting and programming techniques can be used to optimize the computation of the $L_i(\epsilon)$.

**Example 3** For our running example with $\epsilon = 0.3$, it turns out that there are just seven maximal single-processor configurations. These maximal single-processor configurations are shown in Table 1. The numbers in the headings for columns 2-6 are the 5 distinct utilization values in $V(0.3)$, that we determined in Section 4.2 above. Each row corresponds to a different maximal single-processor configuration; it may be verified that the sum of the utilizations in each configuration (i) satisfies Inequality 2 (is no larger than 1.0), and (ii) satisfies Inequality 3 (is at least 0.7, i.e., adding a task with even the smallest utilization would exceed the processor’s capacity). Consider, for example, the single-processor configuration with ID. 2: the sum of the utilizations is $2 \cdot 0.3000 + 1 \cdot 0.3900$, or 0.9900. For the single-processor configuration with ID. 4, the sum of the utilizations is $1 \cdot 0.3000 + 1 \cdot 0.6591$, or 0.9591. ■
4.4 Determining legal multi-processor configurations

We can use the maximal single-processor configurations determined above to determine maximal configurations for a collection of \( m \) processors. Intuitively, each such maximal multiprocessor configuration will represent a different manner in which \( m \) processors can be maximally packed with tasks having utilizations in \( V(\epsilon) \).

**Definition 2 (Multiprocessor configuration)** For given \( m \) and \( \epsilon \), a multiprocessor configuration is an ordered pair of a \( |V(\epsilon)| \)-tuple

\[
\langle y_1, y_2, \ldots, y_{|V(\epsilon)|} \rangle
\]

of non-negative integers, and an \( m \)-tuple

\[
\langle z_1, z_2, \ldots, z_m \rangle
\]

of positive integers \( \leq |L_i(\epsilon)| \). The \( z_j \)'s denote configuration ID’s of single-processor configurations (as previously computed, and stored in \( L_1(\epsilon) \)); \( \langle z_1, z_2, \ldots, z_m \rangle \) thus denotes the \( m \)-processor configuration obtained by configuring the \( j \)’th processor according to the single-processor configuration represented by ID \( z_j \) in \( L_i(\epsilon) \), for \( 1 \leq j \leq m \).

The tuples \( \langle y_1, y_2, \ldots, y_{|V(\epsilon)|} \rangle \) and \( \langle z_1, z_2, \ldots, z_m \rangle \) must satisfy the constraint that for each \( i, 1 \leq i \leq |V(\epsilon)| \), the \( i \)'th component of the tuples in \( L_1(\epsilon) \) with ID’s \( \in \langle z_1, z_2, \ldots, z_m \rangle \) sums to exactly \( y_i \).

A multiprocessor configuration \( \langle \langle y_1, y_2, \ldots, y_{|V(\epsilon)|} \rangle, \langle z_1, z_2, \ldots, z_m \rangle \rangle \) is maximal if there is no other multiprocessor configuration \( \langle \langle y_1', y_2', \ldots, y_{|V(\epsilon)|} \rangle, \langle z_1', z_2', \ldots, z_m' \rangle \rangle \) such that \( y_i \geq y_i' \) for all \( i, 1 \leq i \leq |V(\epsilon)| \).

Let \( L_m(\epsilon) \) denote the list of all maximal multiprocessor configurations. \( L_m(\epsilon) \) can in principle be determined using exhaustive enumeration: simply consider all \( m \)-combinations of the single-processor configurations computed and stored in \( L_1(\epsilon) \). While the worst-case run-time could be as large as \( |L_1(\epsilon)|^m \) and thus once again exponential in \( \epsilon \) and \( m \), this step, like the computation of \( L_1(\epsilon) \), also needs to be performed only once during the process of synthesizing the multiprocessor platform. As was the case with computing \( L_1(\epsilon) \), all manner of counting techniques and programming heuristics may be employed to reduce the run-time in practice. We list a few such optimizations below.

- One obvious such heuristic that can speed up the computation of \( L_m(\epsilon) \) quite significantly is to iteratively compute \( L_j(\epsilon) \) from \( L_{j-1}(\epsilon) \) and \( L_1(\epsilon) \), for \( j = 2, 3, \ldots, m \). In such an iterative approach, only the maximal multiprocessor configurations are retained in each intermediate \( L_j(\epsilon) \). Since the number of maximal multiprocessor configurations in \( L_j(\epsilon) \) is typically far smaller than the worst-case bound of \( |L_1(\epsilon)|^2 \), such an iterative procedure is observed to be far more efficient than determining \( L_m(\epsilon) \) directly from \( L_1(\epsilon) \) by considering all \( m \)-combinations of maximal single-processor combinations.

After it has been computed, \( L_m(\epsilon) \) is stored in a lookup table that is provided along with the \( m \)-processor platform, and is used (in a manner discussed in Section 5 below) for partitioning specific task systems upon the platform.

**Example 4** For our example 4-processor platform with \( \epsilon = 0.3 \), it turns out that there are 140 maximal multiprocessor configurations. Although this is too many to enumerate in this document, we depict a few maximal multiprocessor configurations in Table 2 in the format that they will appear in the lookup table. The numbers in the headings for columns 1-5 are the 5 distinct utilizations; the sixth column lists the 4 maximal single-processor configurations, named according to the configuration ID’s of Table 1, that give rise to this particular maximal multiprocessor configuration.

<table>
<thead>
<tr>
<th>0.3</th>
<th>0.39</th>
<th>0.507</th>
<th>0.6591</th>
<th>0.8568</th>
<th>Single-proc. ID’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>[4 4 5 3]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>[6 5 1]</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>[6 6 7]</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>[7 4 3 2]</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>[4 4 4 3]</td>
</tr>
</tbody>
</table>

**Table 2. Some example maximal 4-processor configurations.**
5 Task assignment

The lookup table of maximal multiprocessor configurations needs to be determined once. Once this lookup table has been obtained, we can use it repeatedly to determine whether any implicit-deadline sporadic task system can be partitioned on this platform. We now describe the partitioning algorithm for doing so.

Let $\tau$ denote a collection of $n$ implicit-deadline sporadic tasks to be partitioned among the (unit-capacity) processors in the $m$-processor platform. Let $u_i$ denote the utilization of the $i$'th task in $\tau$. (I.e., the task system to be partitioned is completely specified by specifying the utilizations of the $n$ tasks in it.) Our task assignment algorithm is depicted in pseudo-code form, in Figure 1. It operates in two phases:

1. In the first phase (Steps 1 and 2 in the pseudo-code), it attempts to assign all tasks with utilization $\geq \epsilon/(1+\epsilon)$. The lookup table constructed as described in Section 4 is used during this phase.

2. Once this phase has been completed, tasks with utilization $< \epsilon/(1+\epsilon)$ are considered during the second phase (Steps 3 and 4 in the pseudo-code). In essence, the algorithm attempts to accommodate those small-utilization tasks in the remaining capacity that is left over in the individual processors after phase 1 is completed.

**Properties.** We will first show that the partitioning algorithm is sound:

**Theorem 1** If the partitioning algorithm of Figure 1 succeeds in assigning all the tasks in $\tau$, then the tasks that are assigned to each processor can be scheduled on that processor to meet all deadlines by preemptive uniprocessor EDF.

**Proof Sketch:** During the first phase (Steps 1 and 2 in the pseudo-code), the algorithm assigns tasks to processors such that the sum of the inflated utilizations of all the tasks on each processor does not exceed the capacity of the processor. Hence, the sum of the original (i.e., non-inflated) utilizations of tasks assigned to any particular processor does not exceed the capacity of the processor. This property is preserved during the second phase of the algorithm (Steps 3 and 4 in the pseudo-code), since a task is only added to a processor during this phase if the sum of the utilizations after doing so will not exceed the processor’s capacity. Hence if the task-assignment algorithm succeeds in assigning all the tasks to processors, then the sum of the utilizations of the tasks assigned to any particular processor is no larger than one. It follows from the optimality of EDF on preemptive uniprocessor platforms [5, 2] that each processor is consequently successfully scheduled by EDF.

And what if the algorithm fails to assign all the tasks in $\tau$ to the processors? In that case, we will now show that no algorithm, not even an optimal one, could have partitioned $\tau$ upon an $m$-processor platform comprised of processors of slightly smaller computing capacity:

**Theorem 2** If the partitioning algorithm of Figure 1 fails to partition the tasks in $\tau$, then no algorithm can partition $\tau$ on a platform of $m$ processors each of computing capacity $1/(1+\epsilon)$.

**Proof Sketch:** The partition algorithm of Figure 1 may declare failure at two points, one of which is in phase one and the other is in phase two. We consider each possible point of failure separately.

1. Suppose that the algorithm reports failure during phase one, while attempting to assign only the tasks with utilization $\geq \epsilon/(1+\epsilon)$ (Step 2 in the pseudo-code). Since each such task has its utilization inflated by a factor $< (1+\epsilon)$, it must be the case that all such (original — i.e., unmodified-utilization) tasks cannot be scheduled by an optimal algorithm on a platform comprised of $m$ processors each of computing capacity $1/(1+\epsilon)$. In other words, even just the tasks in $\tau$ with unmodified utilizations $\geq \epsilon$ cannot be partitioned among $m$ processors of computing capacity $1/(1+\epsilon)$ each, and consequently all of $\tau$ clearly cannot be partitioned on such a platform.

2. Suppose that the algorithm reports failure during phase two, while attempting to assign the tasks with utilization $< \epsilon/(1+\epsilon)$ (Step 4 in the pseudo-code). This would imply that while some task with utilization $< \epsilon/(1+\epsilon)$ remains unallocated to any processor, the sum of the utilizations of the tasks already assigned to each processor is $> (1-\epsilon/(1+\epsilon))$. Therefore the total utilization of $\tau$ exceeds $m \times (1-\epsilon/(1+\epsilon)) = m(1/(1+\epsilon))$, and $\tau$ cannot consequently be feasible on $m$ processors of computing capacity $(1/(1+\epsilon))$ each.

The theorem follows.

**Example 5** Returning to our example ($m = 4$ processors, $\epsilon = 0.3$), let us consider a task system $\tau$ comprised of...
Task system $\tau$, consisting of $n$ implicit-deadline tasks with utilizations $u_1, u_2, \ldots, u_n$, is to be partitioned among $m$ unit-speed processors.

1. For each task with utilization $\geq \epsilon/(1 + \epsilon)$, round up its utilization (if necessary) so that it is equal to $\epsilon \times (1 + \epsilon)^k$ for some non-negative integer $k$. (Observe that such rounding up inflates the utilization of a task by at most a factor $(1 + \epsilon)$: the ratio of the rounded-up utilization to the original utilization of any task is $\leq (1 + \epsilon)$.)

Now all the tasks with (original) utilization $\geq \epsilon/(1 + \epsilon)$ have their utilizations equal to one of the distinct values that were considered during the table-generation step. Let $k_i$ denote the number of tasks with modified utilization equal to $\epsilon \times (1 + \epsilon)^{i-1}$, for each $i$, $1 \leq i \leq |V(\epsilon)|$.

2. Determine whether this collection of modified-utilization tasks can be accommodated in one of the maximal $m$-processor configurations that had been identified during the pre-processing phase. That is, determine whether there is a maximal $m$-processor configuration

$\left(\langle y_1, y_2, \ldots, y_{|V(\epsilon)|}\rangle, \langle z_1, z_2, \ldots, z_m\rangle\right)$

in $L_m(\epsilon)$, satisfying the condition that $y_i \geq k_i$ for each $i$, $1 \leq i \leq |V(\epsilon)|$.

- If the answer here is “no,” then report failure: we are unable to partition $\tau$ among the $m$ processors.
- If the answer is “yes,” however, then a viable partitioning has been found for the tasks with (original) utilization $\geq \epsilon/(1 + \epsilon)$: assign these tasks according to the maximal $m$-processor configuration.

3. It remains to assign the tasks with utilization $< \epsilon/(1 + \epsilon)$. Assign each to any processor upon which it will "fit," i.e., any processor on which the sum of the (original — i.e., unmodified) utilizations of the tasks assigned to the processor would not exceed one if this task were assigned to that processor.

4. If all the tasks with utilization $< \epsilon/(1 + \epsilon)$ are assigned to processors in this manner, then a viable partitioning has been found for all the tasks. However, if some task cannot be assigned in this manner, then report failure: we are unable to partition $\tau$ among the $m$ processors.

Figure 1. Algorithm for partitioning implicit-deadline sporadic tasks on an identical multiprocessor platform scheduled using preemptive partitioned EDF.
tasks with the following utilizations (listed here in non-decreasing order):
\[
\frac{1}{5}, \frac{1}{5}, \frac{1}{3}, \frac{3}{20}, \frac{1}{25}, \frac{5}{5}, \frac{2}{2}, \frac{4}{4}.
\]
Noting that \(\epsilon/(1+\epsilon) = 0.3/1.3 = 0.2308\), we observe that the first two tasks have utilization \(\epsilon/(1+\epsilon)\) and are hence not to be considered during the first steps of the partitioning algorithm.

We round the remaining utilizations up:
\[
\frac{1}{5}, 0.3900, 0.3900, 0.3900, 0.5070, 0.5070, 0.5070, 0.8568.
\]

Using Table 2, we notice that the rounded-up utilizations here correspond to the configuration listed on the third row: \(0 \ 3 \ 3 \ 0 \ 1\), obtained from the single-processor configurations \([6 \ 6 \ 6 \ 7]\) of Table 1.

Accordingly, we assign the tasks with utilization \(\geq \epsilon/(1+\epsilon)\) to the 4 processors as specified in configurations 6, 6, 6, and 7 respectively:

- 6: \(1/3, 2/5\). (Remaining capacity = \(1 - 0.7333 = 0.2667\))
- 6: \(7/20, 1/2\). (Remaining capacity = \(1 - 0.85 = 0.15\))
- 6: \(9/25, 1/2\). (Remaining capacity = \(1 - 0.86 = 0.14\))
- 7: \(3/4\). (Remaining capacity = \(1 - 0.7500 = 0.25\))

It remains to assign the two tasks with utilization \(< \epsilon/(1+\epsilon)\): the ones with utilization \(1/5\) each. These first can be accommodated in the first and last processors, yielding the following mapping:

- 6: \(1/3, 2/5, 1/5\). (Remaining capacity = \(0.0667\))
- 6: \(7/20, 1/2\). (Remaining capacity = \(0.15\))
- 6: \(9/25, 1/2\). (Remaining capacity = \(0.14\))
- 7: \(3/4, 1.5\). (Remaining capacity = \(0.05\))

### 6 Summary

One of the drawbacks of the partitioned approach to multiprocessor real-time scheduling is that partitioning is at least as hard as bin-packing, and is therefore computationally highly intractable (NP-hard in the strong sense). Since exact partitioning is intractable, researchers have considered approximate approaches to this problem. An important result in this regard is the work of Hochbaum and Shmoys [3], which obtained a polynomial-time approximation scheme (PTAS) for the minimum makespan problem.

The minimum makespan problem is essentially equivalent to the problem of partitioning implicit-deadline sporadic task systems under preemptive EDF scheduling; however, the PTAS of [3] does not find much use in partitioned multiprocessor scheduling due to its unacceptably large run-time: \(O(n^2 \log \frac{1}{\epsilon} \log \frac{1}{\epsilon})\). In this research, we have broken up this needed computation into two distinct parts: a very computation-intensive part that must be done only once per platform during platform synthesis time with the results stored in a lookup table for subsequent use, and a more efficiently-implementable part that essentially consists of table lookup and some simple additional processing. An “accuracy” parameter, \(\epsilon\), dictates the size of the lookup table it, and trades this off against the amount of the platform capacity that can be guaranteed usable for scheduling implicit-deadline sporadic task systems.

### References


