Real-Time Syst (2010) 46: 332–359
DOI 10.1007/s11241-010-9109-2

LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines

Shelby Funk

Published online: 29 October 2010

Abstract This article presents a detailed discussion of LRE-TL (Local Remaining Execution-TL-plane), an algorithm that schedules hard real-time periodic and sporadic task sets with unconstrained deadlines on identical multiprocessors. The algorithm builds upon important concepts such as the TL-plane construct used in the development of the LLREF algorithm (Largest Local Remaining Execution First). This article identifies the fundamental TL-plane scheduling principles used in the construction of LLREF. These simple principles are examined, identifying methods of simplifying the algorithm and allowing it to handle a more general task model. For example, we identify the principle that total local utilization can never increase within any TL-plane as long as a minimal number of tasks are executing. This observation leads to a straightforward approach for scheduling task arrivals within a TL-plane. In this manner LRE-TL can schedule sporadic tasks and tasks with unconstrained deadlines. Like LLREF, the LRE-TL scheduling algorithm is optimal for task sets with implicit deadlines. In addition, LRE-TL can schedule task sets with unconstrained deadlines provided they satisfy the density test for multiprocessor systems. While LLREF has a $O(n^2)$ runtime per TL-plane, LRE-TL's runtime is $O(n \log n)$ per TL-plane.

Keywords Multiprocessor scheduling · Hard real-time systems · Periodic tasks · Sporadic tasks · Unconstrained deadlines

1 Introduction

In hard real-time systems, jobs have specific timing requirements, or deadlines, and inability to meet a deadline is considered a system failure. Therefore, we must be able

S. Funk (✉)
University of Georgia, Athens, GA, USA
e-mail: shelby@cs.uga.edu

to determine that no deadlines will be missed before running the system. One way to do this is to use an optimal scheduling algorithm. We say a scheduling algorithm is optimal if it will schedule all jobs to meet their deadlines whenever it is possible to do so. For example, the Earliest Deadline First (EDF) scheduling algorithm (Liu and Layland 1973; Davari and Dhall 1985) is known to be optimal on uniprocessors when preemption is allowed.

As multiprocessors become more popular, they are being used in a wider variety of applications, including real-time and embedded systems. This article examines the problem of scheduling hard real-time tasks on m identical multiprocessors. While there are many advantages to using multiprocessor systems, scheduling with these systems can be complex. For example, even though EDF is optimal on uniprocessors, it is not optimal on multiprocessors. In fact, EDF might miss deadlines on multiprocessors even if processors are idle approximately half the time (Baker 2003, 2005; Phillips et al. 1997).

To date, optimal multiprocessor scheduling algorithms tend to have restrictions that make them less desirable than some non-optimal algorithms. Some common restrictions are

- They have high overhead.
- They apply only to a restrictive model for jobs or processors.
- The schedule must be quantum based (i.e., the scheduler is invoked every q time units for some constant q).

Pfair (Baruah et al. 1996) and LLREF (Cho et al. 2006) (Largest Local Remaining Execution First) are two well known optimal multiprocessor scheduling algorithms. Each suffers from at least one of these shortcomings. While Pfair can schedule both periodic (Liu and Layland 1973) and sporadic (Dertouzos and Mok 1989; Dertouzos 1974) tasks, it applies only to quantum based systems on identical multiprocessors. On the other hand, LLREF is not required to adhere to a quantum based schedule and there is a variation of LLREF called PCG that is optimal on uniform multiprocessor systems (Chen and Hsueh 2008). However, LLREF has high scheduling overhead and applies only to periodic task systems. This article introduces a new scheduling algorithm, LRE-TL, which is based on the LLREF scheduling algorithm. We show LRE-TL is optimal for periodic and sporadic task sets and has much lower scheduling overhead than LLREF.

The remainder of this article is organized as follows. Section 2 provides definitions of terms that will be used throughout subsequent sections. Section 3 provides an overview of the LLREF algorithm, which is used as a starting point for describing LRE-TL. Section 4 describes the LRE-TL scheduling algorithm, proves it is optimal for both periodic and sporadic tasks, and compares it to LLREF. Section 5 discusses how scheduling analysis might be improved for LRE-TL. Finally, Sect. 6 provides some concluding remarks.

2 Model and definitions

This article considers a global multiprocessor scheduling algorithm for periodic (Liu and Layland 1973) and sporadic (Dertouzos and Mok 1989; Dertouzos 1974) task

sets. We assume that tasks are independent and can be preempted at any time. Furthermore, if a task is preempted, it may resume execution on another processor (i.e., migration is also permitted). The overheads associated with these costs are presumed to be built into the task execution times.

A periodic or sporadic task T_i generates a sequence of jobs $T_{i,1}, T_{i,2}, \ldots$. Each task T_i is described using the 3-tuple (p_i, e_i, D_i), where p_i is T_i's period, e_i is its worst case execution time and D_i is its relative deadline. When a job $T_{i,k}$ is released at time $a_{i,k}$ it must be allowed to execute for e_i time units before its deadline $d_{i,k} = a_{i,k} + D_i$. We say T_i is *active* at time t if there exists some $k \geq 0$ such that $t \in [a_{i,k}, d_{i,k})$.

If T_i is a periodic task, then it invokes its first job at time $t = 0$ and all the remaining jobs are invoked exactly p_i time units apart—i.e., $a_{i,k} = (k-1) \cdot p_i$ for all k. If T_i is a sporadic task, then it invokes its first job at any time $t \geq 0$ and the remaining jobs are invoked no less than p_i time units apart—i.e., $a_{i,1} \geq 0$ and $a_{i,k} \geq a_{i,k-1} + p_i$ for all $k > 1$. A task set $\tau = \{T_1, T_2, \ldots, T_n\}$ denotes a set of n periodic or sporadic tasks. Throughout this article, we will clearly state whether τ is assumed to be periodic or sporadic. If $D_i = p_i$, we say T_i has an *implicit deadline*. If $D_i \leq p_i$, we say T_i has a *constrained deadline*, otherwise we say T_i's deadline is *unconstrained*.

One important parameter used to describe a real-time task is its utilization $u_i = e_i/p_i$, which is the proportion of time that T_i must execute between its arrival time and deadline. For sporadic tasks, the utilization measures the "worst-case average demand"—i.e., the average proportion of required computing time assuming the task has a worst case sequence of arrivals ($a_{i,k} = a_{i,k-1} + p_i$). The total utilization of task set τ, denoted $U(\tau)$, is the sum of the individual task utilizations, viz.,

$$U(\tau) = \sum_{i=1}^{n} u_i. \tag{1}$$

One reason why task utilization is such an important parameter is its usefulness in schedulability analysis. For example, a periodic or sporadic task set τ with implicit deadlines can be scheduled to meet all deadlines on m identical processors if and only if the following conditions hold (Baruah et al. 1996; Srinivasan and Anderson 2005)

$$U(\tau) \leq m \quad \text{and} \quad u_i \leq 1 \quad \text{for all } i = 1, 2, \ldots, n. \tag{2}$$

We say any task satisfying (2) is *feasible for m processors*.

When τ contains tasks with constrained deadlines, we use the *density* parameter instead of utilization to analyze the task set's schedulability. A task T_i's density is $\delta_i = e_i / \min\{p_i, D_i\}$, which measures T_i's average demand during its active intervals. The total density of task set τ, denoted $\Delta(\tau)$, is the sum of the individual task utilizations, viz.,

$$\Delta(\tau) = \sum_{i=1}^{n} \delta_i. \tag{3}$$

A task set τ divides the time line into a sequence of intervals, called TL-*planes* (Cho et al. 2006) $[t_{j-1}, t_j)$, where $t_0 = 0$ and for each $j \geq 1$,

$$t_j = \min_{t > t_{j-1}} \{t = d_{i,k} \mid T_i \in \tau, k > 0\}. \tag{4}$$

At each time t within the TL-plane $[t_{j-1}, t_j)$, every task T_i has a *local execution requirement*, $\ell_{i,t}$. This is the amount of time that T_i must execute between time t and time t_j. The progress of each task during the interval $[t_{j-1}, t_j)$ may be viewed as a 2 dimensional plane in which the horizontal axis represents time (T) and the vertical axis represents local execution time (L). Hence, the term TL-plane for these intervals. A task's *local utilization* is the proportion of time T_i must execute during the remainder of the current TL-plane, namely

$$r_{i,t} = \ell_{i,t}/(t_j - t), \tag{5}$$

where t_j is the end of the TL-plane containing t. If $r_{i,t} = 1$ at some point t, then $\ell_{i,t} = t_j - t$. In this case, T_i must execute for the remainder of the TL-plane in order to complete its allocated local execution time. We call such tasks *critical* at time t.

The total local utilization of τ at time t, denoted R_t, is the sum of the individual local utilization values, viz.,

$$R_t = \sum_{i=1}^{n} r_{i,t}. \tag{6}$$

For our discussion, we need to be able to distinguish which tasks have remaining local executing time. We say a task T_i is *operative* at time t if $\ell_{i,t} > 0$ and we let $\mathcal{P}(t)$ be the set of operative tasks.. If a task is not operative, we say it is *stagnant*.

We consider the problem of scheduling n periodic or sporadic tasks on m identical multiprocessors. Without loss of generality, we assume the speed of the processors is 1 for identical multiprocessors—i.e., each processor performs one unit of work per unit of time. Below we discuss the implementation of LLREF on identical multiprocessors. We then examine some of the fundamental underlying attributes of the LLREF scheduling algorithm. We use important observations about these attributes to develop the new algorithm LRE-TL.

3 A brief overview of LLREF

We begin with a description of the LLREF algorithm (Cho et al. 2006), which was proven to be optimal for periodic tasks sets with implicit deadlines (i.e., $D_i = p_i$). Given a task set τ that is feasible for m processors (i.e., τ satisfies the utilization bounds given in (2)), LLREF schedules the tasks so that similar bounds also hold for the τ's *local* utilization—i.e.,

$$\text{For all } t \geq 0 \text{ and } i = 1, 2, \ldots, n,\ R_t \leq m, \quad \text{and} \quad r_{i,t} \leq 1. \tag{7}$$

As stated above, LLREF divides the time horizon into a sequence of consecutive and non-overlapping TL-planes. Each TL-plane ends at some task's deadline and there are no deadlines within any TL-plane. At the beginning of each TL-plane $[t_{j-1}, t_j)$, each task T_i has its local utilization $r_{i,t_{j-1}}$ initialized to u_i. Thus, for every TL-plane $[t_{j-1}, t_j)$,

$$\ell_{i,t_{j-1}} = u_i(t_j - t_{j-1}) \tag{8}$$

Within each TL-plane, LLREF makes scheduling decisions with the aim of achieving the following goals.

1. No processor idles while a job is waiting,
2. No task's local execution becomes negative, and
3. No task's local utilization exceeds 1.

LLREF reschedules the tasks whenever one of these goals is about to be violated. When LLREF invokes a scheduling event, it selects the tasks with the largest local remaining execution and executes them until the next scheduling event. We say a task T_i is *heavier* than another task T_j at a given time t if $\ell_{i,t} > \ell_{j,t}$. Similarly, T_i is *lighter* than T_j at time t if $\ell_{i,t} < \ell_{j,t}$.

At each scheduling point, LLREF executes the heaviest operative tasks. If there are fewer than m operative tasks, then LLREF executes all of the operative tasks. We let x_t denote the maximum number of tasks that can execute simultaneously at a given time t. Because, no more than m tasks can execute at one time, $x_t \leq m$. If there are fewer than m operative tasks, then x_t is the number of operative tasks. Hence,

$$x_t = \min\{m, |\mathcal{P}(t)|\}. \tag{9}$$

LLREF is based on two types of scheduling events. When a scheduling event occurs at time t, LLREF executes the x_t heaviest operative tasks until the next scheduling event.

Assume an event occurs at time t within some TL-plane $[t_{j-1}, t_j)$. Let s denote the latest event (or TL-plane boundary) prior to time t. Then the x_s heaviest tasks execute during $[s, t)$. These tasks execute without interruption until either a bottom (B) event or a critical (C) event occurs or some task has a deadline. Below, we discuss B and C events in more detail.

B (bottom) events occur when a task hits the "bottom" of the TL-plane (i.e., when a task depletes its local remaining execution). Let T_h be the lightest of the x_s tasks that were scheduled to execute at time s. If a B event occurs, it clearly must be triggered by task T_h. Moreover, the B event will occur at time $t = (s + \ell_{h,s})$. If tasks are not rescheduled at this point, then either the processor executing task T_h will become idle, which could cause LLREF's first goal being violated, or T_h's local remaining execution will become negative, which would violate LLREF's second goal.

C (critical) events occur when some task's local utilization becomes 1. Clearly, a C event is caused by the heaviest non-executing task. In particular, if T_h is this heaviest task, then T_h causes a C event at time $t = (t_j - \ell_{h,s})$. If this task is not allowed to execute as soon as the C event occurs, its local utilization will exceed 1. This not only violates the second goal, it also means that T_h will miss its local deadline at time t_j.

The following example illustrates an LLREF schedule.

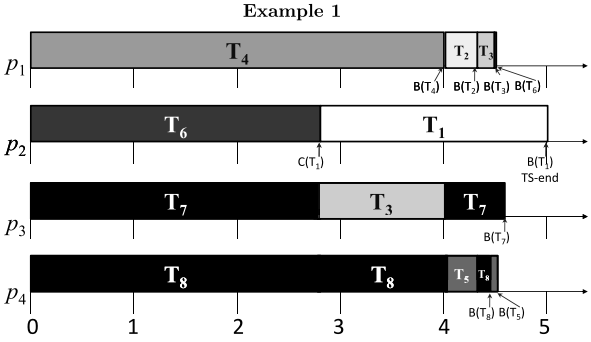
Example 1 We present the LLREF schedule of the task set shown in Table 1, which was used in Cho et al. (2006). We illustrate the schedule on four processors for the first TL plane, $[0, 5)$.

The schedule is shown in Fig. 1. All events are labeled with the event type followed by the task that invoked the event. Each timeline corresponds to one of the four

Table 1 Demonstration task set

	p_i	e_i
T_1	7	3
T_2	16	1
T_3	19	5
T_4	5	4
T_5	26	2
T_6	26	15
T_7	29	20
T_8	17	14

Fig. 1 An LLREF schedule



processors. Initially, the local execution of the tasks (rounded to 2 decimal places) are set as follows.

$$\ell_{1,0} = 2.14, \quad \ell_{2,0} = 0.31, \quad \ell_{3,0} = 1.32, \quad \ell_{4,0} = 4.00,$$
$$\ell_{5,0} = 0.38, \quad \ell_{6,0} = 2.88, \quad \ell_{7,0} = 3.45, \quad \ell_{8,0} = 4.12.$$

At $t = 0$, LLREF executes the 4 heaviest tasks: T_8, T_4, T_7 and T_6. The first possible B event would occur at time 2.88, when T_6 will complete its local execution. The first possible C event would occur at time $5 - 2.14 = 2.86$, when task T_1's local utilization will reach 1. Therefore, the first event within this TL-plane is a C event that occurs at time 2.86 and task T_1 executes for the remainder of the TL-plane.

When the first event occurs, each task's local execution will be as follows.

$$\ell_{1,2.86} = 2.14, \quad \ell_{2,2.86} = 0.31, \quad \ell_{3,2.86} = 1.32, \quad \ell_{4,2.86} = 1.14,$$
$$\ell_{5,2.86} = 0.38, \quad \ell_{6,2.86} = 0.03, \quad \ell_{7,2.86} = 0.59, \quad \ell_{8,2.86} = 1.26.$$

Therefore, tasks T_1, T_3, T_8 and T_4 are selected to execute. The first possible B event would occur at time $2.86 + 1.14 = 4$, when T_4 completes its local execution. The first possible C event would occur at time $5 - 0.59 = 4.41$, when task T_7's local utilization will reach 1. Therefore, the second event within this TL-plane is a B event that occurs at time 4, after which task T_4 will not execute for the remainder of the TL-plane.

Execution will continue in this manner until the TL-plane ends at time 5. Note that some of the execution times are too small to be shown in Fig. 1 (such as the remaining 0.02 units of work on task T_6).

Srinivasan and Anderson (2005), introduced the concept of a *fluid schedule* in which all tasks execute at a constant rate equal to their utilization values. Thus, during an interval $[0, t_0)$, each task T_i will have executed for $u_i \cdot t_0$ time units. Cho et al. (2006), proved that LLREF is optimal by showing that there can never be more than m critical tasks in any TL-plane. Thus, all tasks are able to complete their local execution and LLREF tracks the fluid schedule at TL-plane boundaries (i.e., at every deadline). By definition, if t_j is the deadline of the kth job of some task T_i then $t_j = k \cdot p_i$. Because LLREF tracks the fluid schedule at every deadline, when $t_j = k \cdot p_i$ task T_i will have executed for

$$u_i \cdot (k \cdot p_i) = k \cdot e_i \tag{10}$$

time units. In other words, all jobs will meet their deadlines.

Two shortcomings of LLREF have been noted. First, it incurs fairly high overhead (Kato et al. 2009). Second, it is only defined for periodic tasks with deadlines equal to periods (Devi and Anderson 2010; Cirinei and Baker 2007). Below, we discuss how to address these two shortcomings.

4 LRE-TL: a modification of LLREF

We now present our algorithm, LRE-TL (local remaining execution-TL plane). In Sect. 4.1 below, we present an analytical result that proves LLREF algorithm continues to be optimal for periodic tasks with implicit deadlines even if tasks are not sorted within the TL-planes. We also illustrate an example LRE-TL schedule. In Sect. 4.2, we show how LRE-TL handles sporadic arrivals while still assuming implicit deadlines. In Sect. 4.3 we show how LRE-TL can schedule task sets with unconstrained deadlines (no longer claiming optimality in this case). In Sect. 4.4 we present the LRE-TL algorithm in detail for task sets with unconstrained deadlines and discuss its running time. Finally, we compare the running time of LRE-TL to that of LLREF in Table 2 (Sect. 4.5).

As noted above, LLREF has fairly high running time. When a scheduling event occurs at time t, LLREF must determine which tasks to execute and when the next event will occur. Finally, if $x_t = m$, it must also find when the earliest C event might occur. Thus τ must be at least partially sorted during each scheduling event in order to determine which tasks should be scheduled during the next TL-plane. Maintaining a sorted list is the most expensive portion of each scheduling event, which leads us to question whether this step is actually necessary. Below, we show this sorting step can be skipped and the resulting algorithm is more efficient and can schedule a wider variety of task sets.

4.1 Implicit deadline, periodic tasks

Recall the following result presented by Hong and Leung (1988, 1992)

Theorem 1 (Hong and Leung 1988, 1992) *No optimal online scheduler can exist for a set of jobs with two or more distinct deadlines for any m-processor identical multiprocessor, where* $m > 1$.

This theorem does not claim that no optimal algorithm exists if all deadlines are equal. In fact, Hong and Leung (1988, 1992) present an algorithm, called Reschedule, which is optimal when the jobs all have the same deadline.

LLREF is designed to divide each job into subjobs so that

– The work done by each subjob is proportional to the duration of the TL-plane, and
– At all times *all of the operative tasks have the same deadline.*

The key to our ability to reduce LLREF's running time and to schedule more general task sets is the recognition that we have much more flexibility when deadlines are all shared. We can generalize the concepts behind LLREF and introducing the concept of a *TL-plane-based algorithm*.

Definition 1 Let A be an algorithm that schedules periodic or sporadic task sets on identical multiprocessors. Then A is a *TL-plane-based* algorithm if it adheres to the following rules.

1. Algorithm A begins a new TL-plane at time $t = 0$.
2. If a TL-plane boundary occurs at some time t_{j-1}, then the next TL-plane boundary t_j is set to the earliest deadline $d_{i,k} > t_{j-1}$ of any task T_i that is operative at time t_{j-1}.
3. At the beginning of the TL-plane $[t_{j-1}, t_j)$, the local execution time $\ell_{i,t_{j-1}}$ of each operative task T_i in τ is initialized in proportion to its utilization using (8).
4. Algorithm A schedules each task T_i to execute for $\ell_{i,t_{j-1}}$ time units during the TL-plane $[t_{j-1}, t_j)$.

Clearly, LLREF is one example of a TL-plane-based scheduling algorithm. Below, we show that any TL-plane-based algorithm is optimal for periodic task sets with implicit deadlines. This observation allows us to greatly reduce the running time of LLREF without sacrificing the optimality of the algorithm.

Lemma 1 *Let* τ *be a periodic task set with implicit deadlines. Assume* τ *is scheduled using any TL-plane-based algorithm* A. *Let* $[t_{j-1}, t_j)$ *be some TL-plane. Then the following conditions hold.*

1. *During the TL-plane* $[t_{j-1}, t_j)$, *no task will generate a new job.*

2. *At the beginning of the TL-plane, for each task* T_i *there exists a job* $T_{i,k}$ *that needs to execute for at least* $\ell_{i,t_{j-1}}$ *more time units.*

Proof The proof is by induction on the TL-planes. Because τ is a periodic task set with implicit deadlines, all tasks are operative at time $t_0 = 0$. By the definition of TL-plane-based algorithms

$$t_1 = \min\{p_i \mid T_i \in \tau\}. \tag{11}$$

Therefore, no task will generate a new job before time t_1 and Condition 1 holds. Now consider any task T_i. At time $t = 0$, task T_i generates the job $T_{i,1}$ which needs to execute for e_i time units. By definition,

$$\ell_{i,0} = u_i \cdot (t_1 - t_0) \tag{12}$$

$$\leq u_i \cdot p_i \tag{13}$$

$$= e_i. \tag{14}$$

Thus, both conditions of the lemma hold for the TL-plane $[t_0, t_1)$.

Now, assume the Lemma holds for all TL-planes during the interval $[0, t_{j-1}]$, where t_{j-1} is the start of the jth TL-plane for some $j \geq 1$. Then by the induction hypothesis all tasks are operative at time t_{j-1}. Therefore, t_j is the earliest deadline of all the tasks of τ. Thus, no new jobs can arrive during $[t_{j-1}, t_j)$ and Condition 1 holds for the TL-plane $[t_{j-1}, t_j)$.

We now show that at time t_{j-1} every task T_i has an associated job $T_{i,k}$ that needs to execute for at least $\ell_{i,t_{j-1}}$ more time units. Because A is a TL-plane-based algorithm, task T_i has executed for

$$\sum_{h=1}^{j-1} u_i \cdot (t_h - t_{h-1}) = u_i \cdot t_{j-1} \tag{15}$$

time units during the interval $[0, t_{j-1})$. Let $d_{i,k}$ denote $T_{i,k}$'s deadline. Because A is TL-plane-based and Condition 1 holds for $[t_{j-1}, t_j)$, we know $d_{i,k} \geq t_j$. By the periodicity of T_i

$$k = \left\lceil \frac{t_j}{p_i} \right\rceil. \tag{16}$$

Thus, the total execution time of all tasks released at or before time t_{j-1} is

$$k \cdot e_i = \left\lceil \frac{t_j}{p_i} \right\rceil \cdot e_i \tag{17}$$

$$\geq \frac{t_j}{p_i} \cdot e_i \tag{18}$$

$$= u_i \cdot t_j. \tag{19}$$

Therefore, $T_{i,k}$'s remaining work at time t_{j-1} can be bounded by subtracting (15) from (19), giving a lower bound of

$$u_i \cdot t_j - u_i \cdot t_{j-1} = \ell_{i,t_{j-1}}, \tag{20}$$

so Condition 2 holds for $[t_{j-1}, t_j)$. □

Lemma 1 effectively tells us that as long as we adhere to the TL-plane-based scheduling rules, there will be no surprises within a TL-plane. This predictability is exactly what we need to prove optimality, as the following Corollary illustrates.

Corollary 1 *Let τ be any implicit-deadline periodic task set and let* A *be any TL-plane-based algorithm. Then no jobs will miss their deadlines if algorithm* A *schedules τ.*

Proof Let $T_{i,k}$ be any job and let $d_{i,k}$ denote its deadline. By Lemma 1, $d_{i,k}$ is the end of some TL-plane t_j where

$$t_j = k \cdot p_i. \tag{21}$$

Consider any TL-plane $[t_{h-1}, t_h)$, where $1 \leq h \leq j$. By Lemma 1 task T_i has at least $\ell_{i,t_{h-1}}$ units of outstanding work at time t_{h-1}. Because A is a TL-plane-based algorithm, T_i will execute for exactly $\ell_{i,t_{h-1}}$ time units during $[t_{h-1}, t_h)$. Let $\mathcal{T}_{i,k}$ denote the total time T_i executes during $[0, d_{i,k})$. Then

$$\mathcal{T}_{i,k} = \sum_{h=1}^{j} \ell_{i,t_{h-1}} \tag{22}$$

$$= \sum_{h=1}^{j} u_i \cdot (t_h - t_{h-1}) \tag{23}$$

$$= u_i \cdot t_j. \tag{24}$$

Substituting (21) into (24) gives

$$\mathcal{T}_{i,k} = u_i \cdot (k \cdot p_i) \tag{25}$$

$$= k \cdot e_i. \tag{26}$$

Hence $T_{i,k}$ meets its deadline. □

While Lemma 1 illustrates the promise of using TL-plane-based scheduling algorithms, we still have some difficulty when it comes to creating a TL-plane-based schedule. The first three rules of TL-plane-based scheduling are unambiguous properties that must hold at every TL-plane boundary. The fourth rule is a bit trickier because it only states how long each task should execute without stating when a task should execute.

We now determine some guidelines regarding the choice of tasks to execute at any time s within a TL-plane. Clearly, we can only execute at most m tasks, all of which must be operative, and we must execute all the critical tasks. In addition, if the total local utilization is m we must execute exactly m tasks. Theorem 2 below tells us that these guidelines are adequate. We first show that we can always find a set of tasks adhering to these guidelines if τ's local utilization is feasible for m processors (i.e., if (7) holds for τ).

Lemma 2 *Assume* A *is a TL-plane-based algorithm scheduling some task set* τ. *Let* s *be some time within a TL-plane* $[t_{j-1}, t_j)$ *such that* $R_s \leq m$ *and* $r_{i,s} \leq 1$ *for all tasks* T_i. *Then there exists a set of tasks* X *satisfying the following*:

1. $X \subseteq \mathcal{P}(s)$
2. $\{T_i \mid r_{i,s} = 1\} \subseteq X$
3. $m \geq |X|$
4. *If* $R_s = m$ *then* $|X| = m$.

Proof By Condition 2, X must include all critical tasks (i.e., the tasks T_i such that $r_{i,s} = 1$). Because $R_s \leq m$ we know that there can be no more than m critical tasks. Therefore, it is possible to create a set of operative tasks containing all the critical tasks and no more than m tasks—i.e., a set X exists that satisfies the first three conditions.

Now assume $R_s = m$. By assumption, $0 \leq r_{i,s} \leq 1$ for all tasks T_i. Therefore, $|\mathcal{P}(s) \geq m|$ so a set X of m tasks satisfying the first three conditions exists. □

Recall that LLREF reschedules tasks whenever there is a B or a C event. If an algorithm always selects tasks according to the guidelines given in Lemma 2, we will need one additional event which we call an *F (full) event*.
F (full) events occur when some τ's total local utilization becomes m. Specifically, an F event occurs at any time t_e such that $R_{t_e} = m$ and there exists some $\epsilon > 0$ such that $R_t < m$ for all $t \in [t_e - \epsilon, t_e)$.

We will now prove that A is optimal as long as tasks are selected according to the lemma above and A reschedules the tasks upon every B, C and F event.

Theorem 2 *Let* τ *be a periodic task set with implicit deadlines that is feasible for* m *processors. Let* A *be any scheduling algorithm that adheres to the first three properties of TL-plane-based scheduling* (*Definition* 1). *Assume* A *triggers scheduling events whenever a B, C or F event occurs. Further, assume that when* A *schedules a set of tasks* X *to execute,* X *satisfies the properties of Lemma* 2. *Then* A *will successfully complete all tasks' local execution within every TL-plane. I.e.,* A *adheres to all four properties of TL-plane-based scheduling.*

Proof Let $[t_{j-1}, t_j)$ be any TL-plane. If some task T_i does not execute for $\ell_{i,t_{j-1}}$ time units during the $[t_{j-1}, t_j)$ then there will be some time s, where $t_{j-1} < s < t_j$, such that $r_{i,s} > 1$ and $R_s > m$. Thus, it suffices to prove that for all times s within any TL-plane, $R_s \leq m$ and $0 \leq r_{i,s} \leq 1$ for all tasks T_i. The proof is by induction on the events within the TL-plane.

By assumption, τ is feasible for m processors. At the beginning of each TL-plane $[t_{j-1}, t_j)$, local execution times are set so that $r_{i,t_{j-1}} = u_i$ for all tasks T_i. Therefore, the base case holds.

Now assume A selects a set of tasks X to execute at time s, where $R_s \leq m$ and $0 \leq r_{i,s} \leq 1$ and X satisfies the properties of Lemma 2. Select any time $t_e > s$ such that there are no B, C or F events during the interval $[s, t_e)$. Let v be any time such that $s \leq v \leq t_e$. We wish to show that $R_v \leq m$ and $r_{i,v} \leq 1$ for all tasks T_i.

If $R_s < m$ then $R_v < m$ because no F event can occur between times s and v. If $R_s = m$ then the total remaining execution of all the tasks at time s is $L_s = R_s \cdot (t_j - s)$. By the conditions of Lemma 2, A must execute m tasks during the interval $[s, v)$. Therefore, we can find L_v as follows.

$$L_v = L_s - m \cdot (v - s) \tag{27}$$

$$= m \cdot (t_j - s) - m \cdot (v - s) \tag{28}$$

$$= m \cdot (t_j - v). \tag{29}$$

Thus,

$$R_v = \frac{L_v}{t_j - v} = m. \tag{30}$$

Now consider $r_{i,v}$ for any task T_i. If $0 < r_{i,s} < 1$ then $0 < r_{i,v} < 1$ because there are no B or C events between s and v. If $r_{i,s} = 0$ or $r_{i,s} = 1$ then $r_{i,v} = r_{i,s}$ because A only executes tasks in $\mathcal{P}(s)$ and executes all critical tasks.

We have shown that $R_v \leq m$ and $0 \leq r_{i,v} \leq 1$ for all tasks T_i, where $s \leq v < t_e$. Therefore, $R_{t_e} \leq m$ and $0 \leq r_{i,t_e} \leq 1$ for all tasks T_i. □

Theorem 2 shows that an algorithm can adhere to the TL-plane-based scheduling rules using some fairly straightforward decision mechanisms. With this in mind, we see that many of LLREF's scheduling overheads can be avoided. In particular, there is never any need to sort tasks within a TL-plane and many of the preemptions and migrations within a TL-plan are unnecessary. Of course, if m operative tasks are executing when the C event occurs, then some task must be preempted and that task will have to migrate. Theorem 2 tells us that this is the *only* time such overheads must be incurred within a TL-plane.

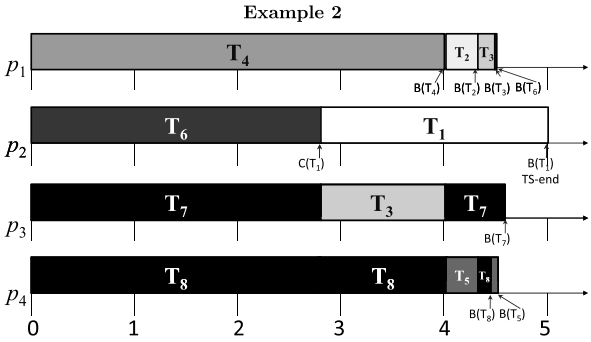
Consider once again the LLREF schedule shown in Fig. 1. There are several points in this schedule where a preemption can clearly be avoided. For example, on processor p_4, task T_5 preempts T_8, which later preempts T_5 and is once again preempted by T_8. This is an artifact of selecting the m tasks with the largest local remaining execution at every scheduling event.

We use the results provided in Theorem 2 to develop algorithm LRE-TL, which is similar to LLREF but has a simpler task selection process within a TL-plane. If a scheduling event occurs at some time t, LRE-TL will schedule x_t tasks (see (9)). Thus, F events will never occur in LRE-TL schedule. Because C events are the ones that cause preemptions and migrations, LRE-TL prefers to schedule heavier tasks. If a preemption is necessary, LRE-TL will preempt the lightest executing task. We illustrate LRE-TL in the following example.

Example 2 Figure 2 illustrates the LRE-TL schedule of the task set described in Table 1, which we used to present the LLREF algorithm in Example 1. Because we have the same task set executing and the mechanism for initializing local execution time is identical, the initial execution times are the same as in Example 1, viz.

$$\ell_{1,0} = 2.14, \qquad \ell_{2,0} = 0.31, \qquad \ell_{3,0} = 1.32, \qquad \ell_{4,0} = 4.00,$$

Fig. 2 An LRE-TL schedule



$$\ell_{5,0} = 0.38, \qquad \ell_{6,0} = 2.88, \qquad \ell_{7,0} = 3.45, \qquad \ell_{8,0} = 4.12.$$

LLREF and LRE-TL both start by selecting the m tasks with the largest remaining execution. Thus, they both initially execute tasks T_8, T_4, T_7 and T_6 to execute at time $t = 0$ until task T_1 has a C event at time 2.86.

At this point the LRE-TL schedule starts to differ from the LLREF schedule. In both schedules, task T_1 will preempt task T_6, which has the lowest remaining execution of all executing tasks, and T_1 will execute for the remainder of the TL-plane. However, in the LRE-TL schedule, the remaining tasks (T_4, T_7 and T_8) will continue to execute. Scheduling will continue in this manner as shown in Fig. 2. Whenever some task T_i incurs a B event, the heaviest waiting will be assigned to the processor that was executing T_i. Whenever a task incurs a C event, it preempts the lightest executing task.

Note that, LRE-TL permits both T_5 and T_8 to execute without being preempted. The only task that is preempted is T_6, which is preempted when T_1 has a C event. At time $t = 0$, the 4 heaviest tasks have local execution larger than $5 - \ell_{6,0} = 2.12$. Therefore, no matter how the TL-plane is scheduled, at least one task must be preempted.

Even for this small example task set, the difference between the two algorithms is quite stark. LLREF triggers 5 preemptions, 2 of which result in a task migration, whereas LRE-TL triggers only 1 preemption and migration[1]. We now examine how to relax the restriction that LRE-TL schedules only strictly periodic tasks with implicit deadlines.

4.2 Scheduling sporadic tasks

Until the introduction of LRE-TL (Funk and Nadadur 2009), there was no known method for modifying LLREF to handle sporadic tasks. The difficulty arose because job arrival times are unknown for sporadic tasks and it was not clear how to handle the arrival of a new job in the middle of a TL-plane. Recall that the optimality of TL-plane-based scheduling algorithms relied on Lemma 1, which states that no task will

[1] Because LRE-TL only preempts tasks when a C event occurs, every preemption will result in a migration. The same holds true for preemptions due to C events in LLREF.

generate a job within a TL-plane. The flexibility we observed in the previous section indicates that handling such arrivals might not be so complicated after all. However, we cannot rely solely on the B, C and F events if jobs may arrive within a TL-plane. If we choose to execute very few of the operative tasks, the total local utilization of the task set may rise over time, in which case a sporadic arrival at some time t might cause R_t to exceed m. Lemma 3 below shows how to prevent total local utilization from increasing over time when scheduling periodic task sets.

Lemma 3 *Assume a TL-plane-based scheduling algorithm* A *is scheduling the implicit-deadline periodic task set* τ *on* m *processors during some TL-plane* $[t_{j-1}, t_j)$. *Let* s *be some time in the TL-plane such that* $R_s \leq m$ *and* $r_{i,s} \leq 1$ *for all tasks* T_i *and assume* A *schedules a set of tasks* X *to execute at time* s *such that*

$$|X| \geq \lfloor R_s \rfloor \tag{31}$$

and let t_e *be the earliest scheduling event that occurs after* s. *Then the total local utilization does not increase during* $[s, t_e)$. *Specifically, for any* $v \in [s, t_e)$

$$R_{t_e} \leq R_v \leq R_s. \tag{32}$$

Proof Because no events occur between s and t_e, the tasks in X must execute during the entire interval $[s, t_e)$. Therefore, the total work done during the interval $[s, v)$ is equal to $(v - s) \cdot |X|$ and we know that

$$\sum_{i=1}^{n} \ell_{i,v} = \sum_{i=1}^{n} \ell_{i,s} - (v - s) \cdot |X|. \tag{33}$$

Therefore, we can determine R_v as follows.

$$R_v = \sum_{i=1}^{n} \frac{\ell_{i,s} - (v - s) \cdot |X|}{t_j - v} \tag{34}$$

$$= \sum_{i=1}^{n} \frac{\ell_{i,s}}{t_j - s} \times \frac{t_j - s}{t_j - v} - \frac{(v - s) \cdot |X|}{t_j - v}. \tag{35}$$

Note that

$$\frac{t_j - s}{t_j - v} = 1 + \frac{v - s}{t_j - v}. \tag{36}$$

Substituting (36) into (35) gives,

$$R_v = \left(R_s + \frac{v - s}{t_j - v} \right) \cdot (R_s - |X|) \tag{37}$$

$$\leq R_s. \tag{38}$$

The last step follows because X contains at least $\lceil R_s \rceil$ tasks and $s < v < t_j$. □

The following observation, which follows directly from Lemma 3, allows us to make TL-plane-based scheduling algorithms more efficient and more flexible.

Observation 1 Let A be a scheduling algorithm that adheres to the first three properties of TL-plane-based scheduling for implicit-deadline periodic task sets and let $X(t) \subseteq \mathcal{P}(t)$ denote the set of tasks A schedules at a given time t. If $|X(t)| \geq R_t$ for all t then total local utilization cannot increase within a TL-plane.

Using this observation, we begin to see how to schedule sporadic task sets with implicit deadlines. First, we must define a new type of scheduling event for task arrivals.

The A Event The A (arrival) event occurs when a sporadic task T_s invokes a new job.

When a task T_s generates a new job at some time t_s in the middle of a TL-plane $[t_{j-1}, t_j)$, we can set T_s's local utilization to be $r_{s,t_s} = u_s$, just as we would at the beginning of a TL-plane. Specifically, we let

$$\ell_{s,t_s} = u_s \cdot (t_j - t_s). \tag{39}$$

Once ℓ_{s,t_s} is determined, the scheduling can proceed as normal if the new job's deadline does not occur within the current TL-plane. If, however, the deadline is before t_j some extra work is required. Theorem 3 demonstrates that any TL-plane-based algorithm can successfully schedule A events when the new job's deadline is not in the current TL-plane.

Theorem 3 *Let τ be an implicit-deadline sporadic task set that is feasible on m processors. Assume τ is scheduled using a TL-plane-based algorithm such that at every scheduling event t_e,* A *executes at least $\lceil R_{t_e} \rceil$ tasks. Assume task T_s generates a new job at some time t_s within TL-plane $[t_{j-1}, t_j)$, where $t_s + p_s \geq t_j$. If* A *sets ℓ_{s,t_s} according to (39), then all tasks will be able to complete their local execution within the current TL-plane and the system will be feasible at time t_j.*

Proof We show that all tasks will be able to complete their local execution by proving that after T_s arrives $R_{t_s} \leq m$ and $r_{i,t_s} \leq 1$ for all T_i. Because T_s generated a new job within the TL-plane, it must have been stagnant at time t_{j-1}. Hence, $R_{t_{j-1}} \leq m - u_s$. By Lemma 3, just prior to t_s we know that $R_{t_s-\epsilon} \leq R_{t_{j-1}}$. Therefore, after letting $r_{s,t_s} = u_s$ (thus, increasing R_{t_s}), we still have $R_{t_s} \leq m$. Also, $r_{i,t_s} \leq 1$ for all tasks T_i because T_s's arrival will not change the utilization of any task other than T_s. Therefore, A will be able to successfully schedule the remainder of the TL-plane.

We now show that the system will be feasible at time t_j. For each active task T_i, let $T_{i,j}$ be the active job at time t_j and let $c_{i,j}$ be the amount of time $T_{i,j}$ has executed since its arrival at time $a_{i,j} \leq t_j$. Because local execution is always set to be proportional to a task's utilization, it must be the case that

$$c_{i,j} = u_i \cdot (t_j - a_{i,j}) \tag{40}$$

$$= u_i \cdot (d_{i,j} - a_{i,j}) - u_i \cdot (d_{i,j} - t_j) \tag{41}$$

$$= e_i - u_i \cdot (d_{i,j} - t_j). \tag{42}$$

Therefore, $T_{i,j}$ will be able to execute for e_i time units by its deadline. □

The correctness of LLREF (and of LRE-TL) hinges on the following two conditions: (i) all tasks complete their local execution by the end of every TL-plane, and (ii) every task's deadline coincides with the end of some TL-plane. These algorithms do not guarantee any timing properties *within* a TL-plane, but they can make guarantees at the boundaries *between* TL-planes.

Because sporadic tasks do not have fixed arrival times, we cannot predict the pattern of the deadlines in advance. Hence, at the beginning of each TL-plane t_{j-1}, we determine the duration of the TL-plane by finding the earliest upcoming deadline, d_{next}, and setting t_j equal to d_{next}. Doing this will ensure that the deadlines of all *operative* tasks will correspond with the end of some TL-plane. If a sporadic task generates a new job with a deadline no earlier than t_j, then Theorem 3 tells us that we can safely schedule all tasks. We now consider jobs arriving with deadlines before t_j. There are three ways to address this possibility.

- We can ensure no sporadic arrival will have a deadline within a TL plane by limiting the length of a TL-plane.
- We can adjust the TL-plane online.
- In some cases, we can ignore the internal deadline by ensuring the sporadic task can execute non-preemptively.

We discuss each of these approaches below.

4.2.1 Avoiding internal deadlines

A sporadic deadline within a TL-plane can be avoided by shortening the TL-plane. As noted above, a task can generate a job within TL-plane $[t_{j-1}, t_j)$ only if it is stagnant at time t_{j-1}. Let $p_{\min}$ be the shortest period of any stagnant task and let d_{next} be the earliest deadline after t_{j-1}. Then no deadline of any operative or stagnant task can occur within the TL-plane if we set t_j, the end of the TL-plane starting at time t_{j-1}, as follows

$$t_j = \min\{d_{\text{next}}, t_{j-1} + p_{\min}\}. \tag{43}$$

While this approach will prevent internal deadlines, it achieves this goal at a cost. Because the resulting TL-plane boundary will not coincide with any task's deadline, we may be unnecessarily increasing the number of TL-plane boundaries. Given that the scheduler's runtime is longest at TL-plane boundaries, we wish to avoid unnecessary boundaries such as this.

4.2.2 Handling internal deadlines online

If a sporadic task T_s generates a job whose deadline within the current TL-plane, we can split the remainder of the TL-plane into two pieces, where the break occurs at T_s's deadline. The remaining execution times of the operative tasks are prorated between the two portions. Specifically, if task T_i is operative at t_s, we split ℓ_{i,t_s} into

two pieces ℓ_{i_1} and ℓ_{i_2}, where

$$\ell_{i_1} = \ell_{i,t_s} \cdot \frac{p_s}{t_j - t_s} \quad \text{and} \tag{44}$$

$$\ell_{i_2} = \ell_{i,t_s} \cdot \left(1 - \frac{p_s}{t_j - t_s}\right). \tag{45}$$

Additionally, we let $\ell_{s,t_s} = e_s$. We then execute two separate TL-planes. The first one executes T_s for e_s time units and each T_i for ℓ_{i_1} time units. The second one executes each T_i for the remaining ℓ_{i_2} time units. By construction, the $\ell_{i,1} = \ell_{i,2}$ for all T_i. Moreover, by Lemma 3, we know that $R_{t_s} \leq m - u_s$ just prior to T_s's arrival. Therefore, the total local utilization will not exceed m in either "sub-plane" and all tasks will have a utilization of at most 1. Hence, any TL-plane-based scheduler can successfully schedule the split tasks. However, this approach takes as long as scheduling a TL-plane boundary. If possible, we would prefer to avoid this scheduling expense.

4.2.3 Ignoring internal deadlines

Because we assume $u_i \leq 1$ for every T_i, we know that if a job is allowed to execute non-preemptively as soon as it arrives, it will meet its deadline. Therefore, when a sporadic task generates a new job with a deadline within the current TL-plane. We can check if the job can execute non-preemptively without causing any deadline misses. By design, LRE-TL will only preempt jobs when a C event occurs. We can ensure that a sporadic task whose deadline is within the current TL-plane will not be preempted provided there won't be too many C events before the task completes execution. When T_s generates a new job at time t_s, it will need to execute for e_s time units. Assume that μ tasks are executing non-preemptively at time t_s. We can ensure T_s can execute without being preempted if there are at most $(m - \mu - 1)$ operative tasks T_i with $\ell_{i,t_s} > t_j - t_s - e_s$. If more than $(m - \mu - 1)$ tasks have larger local execution, we cannot ensure T_s can execute non-preemptively and we will have to split the TL-plane as described above.

Clearly, the approach of ignoring the internal deadline is preferable whenever it can be employed. However, there are times when the internal deadlines cannot be ignored through non-preemptive execution and the TL-plane must be split.

4.3 Unconstrained deadlines

We now extend our analysis to allow for tasks with deadlines not equal to periods. Surprisingly, there are very few changes required to accommodate this model. As indicated in Sect. 2, we use the density rather than the utilization to determine schedulability when deadlines are not equal to periods. Note that it is possible for a feasible task set to have a total density that exceeds m. LRE-TL would not be able to successfully schedule such a system (no TL-plane-based algorithm can). As a result, we do not claim that TL-plane-based scheduling is optimal for task sets with unconstrained deadlines. In fact, Fisher, et al., showed that there are no optimal online algorithms for sporadic task sets with unconstrained deadlines (Fisher et al. 2010).

4.3.1 Constrained deadlines

When $D_i < p_i$, we only need to make two small changes to TL-plane-based scheduling algorithms. First, if T_i is operative at the beginning of some TL-plane, we use the density to determine local execution time

$$\ell_{i,t_{j-1}} = \delta_i \cdot (t_j - t_{j-1}). \tag{46}$$

Second, we use D_i instead of using p_i to determine TL-plane boundaries. Thus, at the beginning of each TL-plane, we continue to determine the duration of the TL-plane by determining the nearest upcoming deadline. However, if we wish to avoid internal deadlines, we need to ensure that

$$t_j \leq t_{j-1} + D_{\min}, \tag{47}$$

where $D_{\min}$ is the shortest relative deadline of all stagnant tasks.

We can easily schedule tasks with constrained deadlines by applying our strategies for schedule sporadic task arrivals. Consider the tasks T_{i_1} and T_{i_2}, where task $T_{i_1} = (p_i, e_i, D_i)$ is a periodic task with $D_i < p_i$, and task $T_2 = (D_i, e_i, D_i)$ is a sporadic task. Because T_{i_2} is a sporadic task with implicit deadlines, we have seen LRE-TL and other TL-plane-based algorithms can schedule this task. Note that LRE-TL can schedule T_{i_2} even if it arrives periodically with a period of p_i—i.e., even if its arrival patterns exactly mimics T_{i_1}. Therefore, LRE-TL can schedule T_{i_1}, or any task with a constrained deadline, without any further changes.

4.3.2 Unconstrained deadlines

When $D_i > p_i$, a task could have several jobs within a single TL-plane. When this occurs, we can run into two difficulties. First, a task might generate a new job before the previous job has completed execution. In this case, we would execute the jobs in FIFO order as usual—the job that was released earlier must finish executing before the next job can begin. The local execution requirement $\ell_{i,t}$ is always applied to whichever job of T_i arrived earliest. A more difficult issue arises, however, when the current job finishes executing before the next job arrives. Clearly, we cannot execute T_i in this case even though $\ell_{i,t} > 0$—i.e., T_i is operative, but cannot execute. If there are at least x_t jobs with remaining work, there will be no problem. However, if local utilization can be positive for some tasks that have no work it is possible to have fewer than x_t operative tasks at some time t. Thus, we cannot adhere to the requirement that x_t tasks are always executing. In order to avoid this issue, we use $\min\{p_i, D_i\}$ to determine TL-plane boundaries. Using this approach ensures that only one job of each task T_i will execute within any TL-plane even when $D_i > p_i$.

In this section, we have shown how to schedule sporadic tasks with unconstrained deadlines using a TL-plane-based scheduling such as LRE-TL. We now describe the LRE-TL algorithm in more detail.

4.4 Algorithm LRE-TL-main

The LRE-TL algorithm is comprised of four procedures. The main procedure determines which type of events have occurred, calls the handlers for those events, and instructs the processors to execute their designated tasks. At each TL-plane boundary, LRE-TL calls the TL-plane initializer. Within a TL-plane, LRE-TL processes any A, B or C events (LRE-TL does not have to consider F events because it always executes at least x_t tasks). The TL-plane initializer sets all parameters for the new TL-plane. The A event handler determines the local remaining execution of a newly arrived sporadic task. The B and C event handler preempts one or more tasks if necessary and assigns new tasks to execute. In this section, we ensure that deadlines cannot occur within TL-planes. In particular, we ensure no TL-planes are longer than $q_{\min}$ time units, where $q_{\min} = \min\{D_i, p_i\}$ for all stagnant tasks T_i. At the end of the section, we discuss how the procedures should be modified to allow for interior deadlines.

Throughout this section, we discuss executing *tasks* rather than executing *jobs*. Because we use $\min\{D_i, p_i\}$ to determine TL-plane boundaries, at most one job of T_i overlaps with any TL-plane. Thus, there is no confusion about which job of each task is being scheduled—we always schedule the job that overlaps with the current TL-plane.

LRE-TL maintains all operative tasks in one of two heaps: H_B contains the tasks that are currently executing, and H_C contains the tasks that are waiting to execute. For both heaps, when a task is added to the heap, its key is set to the time at which the task will trigger a scheduling event. Heap H_B contains the set of executing tasks. Task $T_i \in H_B$ triggers a B event if it executes until time $(t + \ell_{i,t})$ without interruption, where t is the current time. Therefore, H_B is a min heap whose key is $(t + \ell_{i,t})$. Heap H_C is the set of operative tasks that are not executing (i.e., the tasks with $\ell_{i,t} > 0$ that are not assigned to any processor at time t). Task $T_i \in H_C$ triggers a C event if it does not execute before time $(t_j - \ell_{i,t})$, where t_j denotes the end of the current TL-plane. Therefore, H_C is a min heap whose key is $(t_j - \ell_{i,t})$. In addition, the heap H_D is used to keep track of all upcoming deadlines. Its key is $a_i + \min\{D_i, p_i\}$, where a_i is time T_i most recently generated a job. Finally, H_S keeps track of all stagnant tasks to ensure no tasks arrive in a TL-planes with an internal deadline. It is a min heap whose key is $\min\{D_i, p_i\}$.

Each task has two fields. T_i.key is the time when T_i will cause an event (the event type depends on which heap T_i is in). T_i.proc-id is the processor T_i should execute on and is valid only if T_i is in H_B. In addition, each processor z has one field, z.task-id, which is the task currently assigned to processor z.

A simplifying observation Note that an operative task T_i's key does not change as long as T_i remains in the same heap. If $T_i \in H_C$, then $\ell_{i,t}$ is not changing over time, so $(t_j - \ell_{i,t})$ remains constant. Also, if $T_i \in H_B$, then $\ell_{i,t}$ decreases as t increases, so $(t + \ell_{i,t})$ remains constant. If a task T_i switches from one heap to the other at time t, then its new key is set to $(t_j - T_i.\text{key} + t)$, where T_i.key was its key prior to switching heaps. Thus, we can maintain proper execution without having to update ℓ at each B or C event.

Algorithm 1 LRE-TL-Main

```
1:  if t_cur = t_j then
2:      TL-Plane-Initialize
3:  else
4:      if an A event occurred then
5:          LRE-TL-A-Event
6:      if a B or C even occurred then
7:          LRE-TL-BC-Event
8:  if H_B.size()> 0 then
9:      t_next ← H_B.min-key()
10:     if H_C.size()> 0 then
11:         t_next ← min{t_next, H_C.min-key()}
12:     if H_S.size()> 0 then
13:         t_next ← min{t_next, H_S.min-key()}
14: else
15:     t_next ← t_j
16: let each processor execute its designated task
17: sleep until time t_next
```

The heaps have five methods. H.min-key() returns the value of the H's minimum key. H.size() returns the number of objects in H. H.extract-min() removes the object with the smallest key from the heap H. H.insert(I) inserts item I into the heap. H.find-key(k) returns a pointer to the object whose key equals k if one exists and returns NULL otherwise. The first two methods run in $O(1)$ time and the last three run in $O(\log H.\text{size}())$ time.

LRE-TL's main algorithm is illustrated in Algorithm 1. At the beginning of a TL-plane, LRE-TL will initialized the TL-plane. Within a TL-plane, it will process all A, B and C events. Once the initializer or events are completed, LRE-TL instructs the processors to execute their designated tasks and sleeps until the next event occurs.

The TL-plane initializer is illustrated in Algorithm 2. It first finds t_j (lines 1 through 7), which is set to the earliest upcoming deadline, but is never larger than $(t_{\text{cur}} + q_{\min})$, where $q_{\min}$ is the earliest TL-plane boundary that a stagnant task might invoke. Once t_j is identified, the local execution values of operative tasks are initialized accordingly (lines 9 through 20). The first m tasks are inserted into H_B and the remaining tasks are inserted into H_C. The keys are set to the time when the task will trigger a B event (if the task is in H_B) or a C event (if the task is in H_C). If there are fewer than m operative tasks, the idle processors' task id's are set to NULL (lines 23 through 24). In order to ensure LRE-TL initially assigns the heaviest tasks to execute, we require that tasks are initially sorted by their density.

The A event handler is shown in Algorithm 3. When a sporadic task T_s arrives, this algorithm determines T_s's local remaining execution, ℓ, and adds T_s to one of the heaps (H_B or H_C). If some processor is idle, then T_s is added to H_B (lines 3 through 6). If there are m operative tasks, then T_s will only preempt a task if its has zero laxity. Hence, when $H_B.\text{size}() = m$, T_s is added to H_C if $u_s < 1$(lines 8 through 10) and T_s is added to H_B and the lightest executing task T_b is moved from

Algorithm 2 TL-Plane-Initialize

Require: $\mathcal{C}$ contains the set of all operative tasks. H_B and H_C are both empty. τ is sorted by density.

```
 1: for all tasks T_i that arrived at time t_cur do
 2:     H_D.insert(t_cur + min{p_i, D_i})
 3:     if T_i ∈ H.S then
 4:         delete T_i from H_S
 5: t_j ← t_cur + H_S.find-min()
 6: if H_D.min-key() ≤ t_j then
 7:     t_j ← H_D.extract-min()
 8: z ← 1
 9: for all i = 1 to m do
10:     if T_i ∈ C then
11:         ℓ ← u_i(t_j − t_cur)
12:         if z ≤ m then
13:             T_i.key ← t_cur + ℓ
14:             T_i.proc-id ← z
15:             z.task-id ← T_i
16:             H_B.insert(T_i)
17:             z ← z + 1
18:         else
19:             T_i.key ← t_j − ℓ
20:             H_C.insert(T_i)
21:     else
22:         H_S.insert(min{D_i, p_i})
23: for all processors z′ s.t. m ≥ z′ > z do
24:     z′.task-id ← NULL
```

H_B to H_C if $u_s = 1$ (lines 12 through 19). Before returning, T_s is inserted into H_D and removed from H_S (lines 20 through 22).

The B and C event handler is shown in Algorithm 4. It identifies which task(s) caused the events. After this algorithm executes the following conditions hold: (i) x_{t_j} tasks are executing, (ii) $\ell_b > 0$ for all $T_b \in H_B$ and (iii) $r_c < 1$ for all $T_c \in H_C$. The algorithm begins by handling any B events (lines 1 through 11). Any tasks $T_b \in H_B$ with remaining execution equal to 0 are removed from H_B and replaced by a waiting task (if one exists). The algorithm then handles the C events (lines 12 through 22). Any tasks $T_c \in H_C$ with utilization equal to 1 are removed from H_C and swapped with some executing task T_b (lines 14 through 9).

4.4.1 Interior deadlines

The procedures presented above prevent jobs that arrive within a TL-plane from having a deadline within the TL-plane. If deadlines can occur within a TL-plane a few changes would be required. First, the length of the TL-plane would be determined

Algorithm 3 LRE-TL-A-Event

Require: The sporadic task T_s that invokes a job to trigger this algorithm is not in $\mathcal{C}$.

```
1:  ℓ ← u_s(t_j − t_cur)
2:  if H_B.size() < m then
3:      T_s.key ← t_cur + ℓ
4:      T_s.proc-id ← z{z is any idling processor}
5:      z.task-id ← T_s
6:      H_B.insert(T_s)
7:  else
8:      if u_s < 1 then
9:          T_s.key ← t_j − ℓ
10:         H_C.insert(T_s)
11:     else
12:         T_b ← H_B.extract-min()
13:         T_s.key ← t_cur + ℓ
14:         T_b.key ← t_j − T_b.key+t_cur
15:         z ← T_b.proc-id
16:         T_s.proc-id ← z
17:         z.task-id ← T_s
18:         H_B.insert(T_s)
19:         H_C.insert(T_b)
20: if H_D.find-key(t_cur + min{D_s, p_s}) = NULL then
21:     H_D.insert(t_cur + min{D_s, p_s})
22: H_S.delete(T_i)
```

only by H_D.min-key() and not by $q_{\min}$. Thus, lines 5 and 6 would be removed from Algorithm 3.

If we wish internal deadlines to always cause the TL-plane to be split, then lines 8 through 19 of Algorithm 3 should only be executed if $t_{cur} + \min\{D_s, p_s\} \leq t_j$. Otherwise, the prorated execution times ℓ_{i_1} and ℓ_{i_2} must be determined for all operative tasks T_i. The current keys are replaced with the first values and ℓ_{s,t_s} is set to e_s. Execution proceeds as normal until the split point, at which time the keys are replaced with the second ℓ values.

If we wish to avoid splitting the TL-planes by using the non-preemptive approach, we maintain the non-preemptable tasks in a new heap H_N. Tasks in H_N can invoke B events, so their key is the local remaining execution time. However, these task are not considered for preemption if a C event occurs. When a task with an internal deadline arrives, before we split the TL-plane, we find the lightest preemptable executing task T_x and check if $t_j - t_s - e_s$ is larger than the remaining execution time of $(m - \mu)$ tasks in $H.C \cup \{T_x\}$, where $\mu = H_N$.heap-size(). If so, the TL-plane must be split as described above. Otherwise, we let T_s preempt T_x and assign T_s to execute. Because T_s is non-preemptable, it is added to H_N instead of H_B.

Algorithm 4 LRE-TL-BC-Event

Require: Each task $T_b \in H_B$ is executing on processor T_b.proc-id and will cause a B event at time T_b.key. Each task $T_c \in H_C$ has positive local execution, is not executing and will cause a C event at time T_c.key. The current time is t_{cur} and the current TL-plane ends at time t_j.

```
1:  while H_B.min − key() = t_cur do
2:      T_b ← H_B.extract-min()
3:      z ← T_b.proc-id
4:      if H_C.size()> 0 then
5:          T_c ← H_C.extract-min()
6:          T_c.key ← t_j − T_c.key+t_cur
7:          T_c.proc-id ← z
8:          z.task-id ← T_c
9:          H_B.insert(T_c)
10:     else
11:         z.task-id ← NULL
12: if H_C.size()> 0 then
13:     while H_C.min-key()= t_cur do
14:         T_b ← H_B.extract-min()
15:         T_c ← H_C.extract-min()
16:         T_b.key ← t_j − T_b.key+t_cur
17:         T_c.key ← t_j − T_c.key+t_cur
18:         z ← T_b.proc-id
19:         T_c.proc-id ← z
20:         z.task-id ← T_c
21:         H_B.insert(T_c)
22:         H_C.insert(T_b)
```

Table 2 Total run time per TL-plane for the scheduling events when executing LRE-TL and LLREF

Running time	LLREF	LRE-TL
Initialize TL-plane	$O(n)$	$O(n)$
A events (per event)	NA	$O(\log n)$
B&C events (per TL-plane)	$O(n^2)$	$O(n \log n)$

4.5 Run time analysis

The running time of LRE-TL depends on what type of event is performed (i.e., Initialization, A, B or C). Table 2 presents LLREF's running time per TL-plane for each type of event handler. When applicable, LLREF's running time is also presented.

The TL-plane initializer has a loop that iterates $O(n)$ times. During these iterations, the two heaps H_B and H_C are populated. Because we assume the tasks of τ are sorted by their density, we populate the heaps in sorted order, which can be done in

$O(n)$ time. LLREF's initializer operates in a similar manner and also has a run time of $O(n)$.

The A event handler initializes the sporadic task parameters, inserts the sporadic task (and possibly one other task) into one of the task heaps and adds the deadline to the deadline heap. Because the deadline heap contains the deadline of every task in the B and C heaps, the running time of the A event handler $O(\log H_D.\text{size}()) = O(\log n)$. As LLREF does not handle sporadic tasks, LRE-TL's running time for handling sporadic arrivals cannot be compared to LLREF.

The running time of the B and C event handler depends on the number of operative tasks, which we will denote α. If $\alpha > m$, the handler moves tasks between the two heaps, which takes $O(\log m + \log(\alpha - m))$ time. Otherwise, H_C is empty and a task is simply removed from H_B, which takes $O(\log \alpha)$ time. Each task causes at most one B or C event per TL-plane. Hence, the total running time within any TL-plane is

$$\sum_{\alpha=1}^{m} \log \alpha + \sum_{\alpha=m+1}^{n} (\log m + \log(\alpha - m)) \tag{48}$$

$$\leq \sum_{\alpha=1}^{n} \log m + \sum_{\alpha=m+1}^{n} \log(\alpha - m) \tag{49}$$

$$= O(n \log m + (n - m) \log(n - m)) \tag{50}$$

$$= O(n \log n). \tag{51}$$

By contrast, LLREF needs to identify the x_t tasks with the largest local remaining execution time at every scheduling event. Therefore, the tasks must be sorted at each B or C event. If the tasks are sorted at the beginning of the TL-plane, the process of re-sorting during a scheduling event can be done in $O(n)$ time by using the prior sort order. The tasks that execute during $[s, t)$ all have their local execution reduced by $(t - s)$ and the ones that did not execute all have unchanged local execution. Thus, it is easy to see that these tasks will maintain the same relative order at time t. Hence, the proper sorting order for the tasks at time t can be found by merging these two sets of task in $O(n)$ time. As stated above, there are at most n events in a TL-plane. Thus, the total time per TL-plane that LLREF spends handling B and C events is $O(n^2)$. Thus, we see that the running time within each TL-plane of LLREF and LRETL is $O(n^2)$ and $O(n \log n)$, respectively.

There is one benefit of LRE-TL that is not captured in the discussion of running time—namely, the reduction in the number of preemptions and migrations. For both LLREF and LRE-TL, there can be at most $O(m)$ C events. In LRE-TL, each C event causes a single preemption and a single migration. The A and B events cause no preemptions or migrations. Therefore, the maximum number of preemptions and migrations per TL-plane is $O(m)$. On the other hand, LLREF sorts the tasks upon every B or C event. Thus, each event could cause m tasks to be preempted and each of these tasks might resume execution a different processor. Thus, we can have $O(m)$ preemptions and migrations per event. Because each task can cause at most one event, the maximum number of preemptions and migrations within each TL-plane is $O(mn)$. The numbers of preemptions and migrations per TL-plane for LLREF and LRE-TL are shown in Table 3.

Table 3 Comparison of preemptions and migrations per TL-plane

Other overhead	LLREF	LRE-TL
Number of preemptions	$O(mn)$	$O(m)$
Number of migrations	$O(mn)$	$O(m)$

5 Schedulability Analysis

Using the above analysis, we can find a bound for the maximum scheduling overhead per TL-plane for LRE-TL schedules of implicit-deadline periodic tasks. We can build this overhead the into schedulability test given in (2). By our assumption we know that no arrivals can occur within a TL-plane. Therefore, we can find an upper bound on the number of events within a TL-plane as follows.

- The initialization procedure executes once.
- The maximum number of C events is $n_c \leq (m-1)$.
- The maximum number of B events is $n_b \leq (n-n_c)$.

B and C events can be processed locally while the other processors continue executing and initialization impacts all m processors. Also, because C events involve two tasks and invoke a preemption, they are more costly than B events. Therefore, if C_X is the scheduling time required for procedure X, the maximum scheduling cost within a TL-plane is at most

$$m \cdot (C_{\text{init}} + C_C) + (n-m) \cdot C_B. \tag{52}$$

Now consider the overheads due to context switches, preemption and migration. If all tasks run to their full worst-case execution time, then each task T_i will execute for exactly $\ell_{i,t_{j-1}}$ time units during each TL-plane $[t_{j-1}, t_j)$. Thus, each TL-plane schedule will be a scaled version of every other TL-plane schedule. In particular, at most $(m-1)$ tasks will migrate at the beginning of each TL-plane and at most $(m-1)$ tasks will migrate within each TL-plane. Similarly, at most $(m-1)$ tasks will be loaded twice within a TL-plane and the remaining tasks will be loaded once. Thus, we can bound these overheads as follows

$$C_{cs} \cdot (n+m-1) + C_{\text{preempt}} \cdot (m-1) + 2C_{\text{migr}} \cdot (m-1), \tag{53}$$

where C_{cs}, C_{preempt} and C_{migr} are the costs of context switching, preemption and migration, respectively.

Equations (52) and (53) can be added together to derive the worst case overhead within a TL-plane. However, this gives a gross over estimate. For periodic task sets with implicit deadlines, a better approach would be to observe a single TL-plane and use that as a template for all the TL-planes. For example, in Fig. 2 we see that T_6 migrates twice—once at the beginning of the TL-plane and once after it is preempted. All the other tasks execute once without migrating. Assuming the tasks execute on 4 processors, this gives a total cost of

$$4C_{\text{init}} + C_C + 7C_B + 9C_{\text{cs}} + C_{\text{preempt}} + 2C_{\text{migr}}. \tag{54}$$

This is a great savings compared to using (52) and (53), which would give

$$4(C_{\text{init}} + C_C) + 4C_B + 11C_{\text{cs}} + 3C_{\text{preempt}} + 6C_{\text{migr}}. \tag{55}$$

Further improvements can be developed though careful scheduling. For example, if T_6 starts migrating whenever it stops executing, then the processor will not have to wait when it is ready to execute the task. This will not interfere with the other tasks because migration can be processed while all four tasks are executing. This strategy will only work, though, if we know the task's migration pattern.

In addition to knowing the overhead per TL-plane, we need to know how many TL-planes can impact each task. This can be determined in pseudo-polynomial time by calculating the arrivals and deadlines within a hyperperiod.

For each T_i, we let κ_i be the maximum number of TL-planes that overlap with any single job of T_i and let v be the maximum overhead during any individual TL-plane. If we penalize all tasks by (v/n) within each TL-plane, then for each T_i, the modified utilization would be

$$u_i' = (e_i + \kappa_i(v/n))/p_i \quad \text{or} \quad u_i' = u_i + (\kappa_i/p_i)(v/n). \tag{56}$$

Then τ is LRE-TL schedulable on m processors if

$$u'_{\max} \leq 1 \text{ and } U'(\tau) \leq m, \tag{57}$$

where $u'_{\max}$ and $U'(\tau)$ are the maximum and total values of u_i' over all tasks T_i.

The above analysis can be modified to account for periodic tasks with unconstrained deadlines. If the scheduler prevents interior task arrivals, then the above analysis works without modification. Otherwise, A events must added onto the determination of v. Handling the overhead for sporadic tasks is much more challenging because κ_i is unknown. The worst case estimate would assume that all tasks arrive as often as possible and no arrivals coincide. This estimation would in all likelihood grossly overstate the overhead, resulting in a virtually unusable test.

Currently, worst-case execution times are assumed to be negligible or built into the worst case execution times. Thus, the schedulability tests are either unrealistic or they are overly pessimistic. The test presented in this section is the first step in trying to find some more useful point between our two current methods. While it is still quite pessimistic, it does provide some insight into how schedulability overhead can be separated from the schedulability test.

6 Conclusion

This article presents the LRE-TL scheduling algorithm, a variation of the LLREF scheduling algorithm that is able to schedule sporadic tasks as well as tasks with unconstrained deadlines. LRE-TL is optimal for task systems with implicit deadlines. In the process of developing LRE-TL, some of the underlying TL-plane-based principles behind LLREF are explored in detail. In particular, LLREF breaks jobs into subjobs that share the same deadlines. When deadlines are all the same, multiprocessor scheduling algorithms have much more flexibility. LRE-TL is able to exploit that

flexibility in a manner that reduces the algorithm runtime from $O(n^2)$ to $O(n \log n)$. Furthermore, LRE-TL reduces the number of preemptions and migrations per TL-plane from $O(mn)$ to $O(m)$. Finally, we have briefly presented a method of incorporating the scheduling overhead into a feasibility test. Several tools for improving the efficiency of the scheduling algorithms and schedulability tests are presented in the development of these results. We plan to continue in this vein in the future.

Both LLREF and LRE-TL achieve optimality by ensuring all tasks track the fluid schedule at TL-plane boundaries. We realize that this is an overconstraint. Tasks may still be able to meet their deadlines even if they diverge from the fluid schedule at some of the TL-plane boundaries. They only really need to track with the fluid schedule at the TL-plane boundaries that coincide with their own deadlines. In future, we plan to explore methods of allowing tasks to diverge from the fluid schedule without causing any deadline misses. Our goal is to further reduce the number of preemptions and migrations. Such methods would have to be heuristic in nature because minimizing the number of preemptions and migrations of a periodic or sporadic task set is an NP-complete problem closely related to the bin packing problem.

We would also like to examine LRE-TL in the uniform multiprocessor context. A uniform multiprocessor variant of LLREF called PCG has already been developed for implicit-deadline periodic (Chen and Hsueh 2008). We plan to examine if the techniques presented in this article can also apply to PCG, allowing sporadic tasks with unconstrained deadlines to be scheduled as well. We suspect that the techniques employed in developing LRE-TL can be applied in a number of useful contexts.

Finally, the incorporation of overheads into the schedulability test has great potential to improve schedulability analysis. We would like to explore the strategy of separating schedulability overhead from the schedulability analysis more fully. Doing so will give us a more effective way of comparing algorithms than the simple comparison of utilization bounds.

References

Baker T (2003) Multiprocessor EDF and deadline monotonic schedulability analysis. In: 24th real-time systems symposium

Baker T (2005) An analysis of EDF schedulability on a multiprocessor. IEEE Trans Parallel Distrib Syst 16(8):760–768

Baruah SK, Cohen N, Plaxton CG, Varvel D (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica 15(6):600–625

Chen SY, Hsueh CW (2008) Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. In: Proceedings of the 2008 real-time systems symposium, pp 147–156

Cho H, Ravindran B, Jensen ED (2006) An optimal real-time scheduling algorithm for multiprocessors. In: Proceedings the 27th IEEE real-time system symposium (RTSS). IEEE Comput. Sci., Los Alamitos, pp 101–110

Cirinei M, Baker T (2007) EDZL scheduling analysis. In: Euromicro conference on real-time systems. ECRTS, pp 9–18.

Davari S, Dhall SK (1985) On a real-time task allocation problem. In: Proceedings of the international conference on system science, pp 133–141

Dertouzos M (1974) Control robotics: the procedural control of physical processors. In: Proceedings of the IFIP congress, pp 807–813

Dertouzos M, Mok AK (1989) Multiprocessor scheduling in a hard real-time environment. IEEE Trans Softw Eng 15(12):1497–1506

Devi UC, Anderson JH (2010) A schedulable utilization bound for the multiprocessor pfair algorithm. Real-Time Syst 38(3):237–288
Fisher N, Goossens J, Baruah S (2010) Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. Real-Time Syst 45(1):26–71
Funk S, Nadadur V (2009) LRE-TL: An optimal multiprocessor algorithm for sporadic task sets. In: International conference on real-time and network systems (RTNS), Paris, France, pp 159–168
Hong KS, Leung JYT (1988) On-line scheduling of real-time tasks. In: Proceedings of the real-time systems symposium, pp 244–250
Hong KS, Leung JYT (1992) On-line scheduling of real-time tasks. IEEE Trans Comput 41(10):1326–1331
Kato S, Yamasaki N, Ishikawa Y (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: Euromicro Conference onReal-Time Systems (ECRTS), Ireland, Dublin, pp 249–258
Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J ACM 20(1):46–61
Phillips CA, Stein C, Torng E, Wein J (1997) Optimal time-critical scheduling via resource augmentation. In: Proceedings of the twenty-ninth annual acm symposium on theory of computing. El Paso, Texas, pp 140–149
Srinivasan A, Anderson JH (2005) Fair scheduling of dynamic task systems on multiprocessors. J Syst Softw 77(1):67–80

Shelby Funk is an assistant professor at the University of Georgia. She received her Ph.D. from the University of North Carolina at Chapel Hill in 2004. Her research interests include multiprocessor real-time scheduling theory, scheduling algorithms, power-aware scheduling and algorithms for uniform multiprocessors.