LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets

Shelby Funk, Vijaykant Nanadur

In Proc. of the 17th International Conference on Real-Time and Network Systems
RTNS'2009, Paris, ECE, 26-27 October, 2009

LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets

Shelby Funk and Vijaykant Nadadur
University of Georgia
Athens, GA, USA
{*shelby,nadadur*}*@cs.uga.edu*

Abstract

This paper introduces LRE-TL, a scheduling algorithm based on LLREF, and demonstrates its flexibility and improved running time. Unlike LLREF, LRE-TL is optimal for sporadic task sets. While most LLREF events take $O(n)$ time to run, the corresponding LRE-TL events take $O(\log n)$ time. LRE-TL also reduces the number of task preemptions and migrations by a factor of n. Both identical and uniform multiprocessors are considered.

1. Introduction

In hard real-time systems, jobs have specific timing requirements, or deadlines. In these systems, inability to meet a deadline is considered a system failure. Therefore, it must be known 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 [1], [2] is known to be optimal on uniprocessors when preemption is allowed.

As multiprocessors become more popular, they are used in a wider variety of applications, including real-time and embedded systems. This paper considers scheduling on identical multiprocessors, which contain m identical processors, and on uniform multiprocessors, which contain m processors whose speeds may vary from one another. While there are many advantages to using multiprocessor systems, scheduling with these systems can be complex. For example, on multiprocessors EDF is not optimal. In fact, EDF might miss deadlines on multiprocessors even if processors are idle approximately half the time [3], [4], [5].

To date, optimal multiprocessor scheduling algorithms tend to have restrictions that make them less desirable than other non-optimal algorithms. Some common restrictions are that (i) they have high overhead, (ii) they apply only to a restrictive model for jobs or processors, or (iii) the schedule must be quantum based (i.e., the scheduler is invoked every q time units for some constant q). There are two well known optimal multiprocessor scheduling algorithms, Pfair [6] and LLREF [7], each suffer from at least one of these shortcomings. While Pfair can schedule both periodic [1] and sporadic [8], [9] tasks, it applies only to quantum based systems on identical multiprocessors. On the other hand, LLREF can be scheduled on both identical and uniform multiprocessors, but it has high scheduling overhead and applies only to periodic task systems. This paper 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. Table 1 illustrates the running time of LRE-TL compared to LLREF (details provided in Section 4.4).

The remainder of this paper is organized as follows. Section 2 provides definitions of terms that will be used throughout the remainder of the paper. 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 compares an LLREF schedule and an LRE-TL schedule. Section 6 discusses how to schedule LRE-TL on uniform multiprocessors. Finally, Section 7 provides some concluding remarks.

2. Model and Definitions

This paper considers a global multiprocessor scheduling algorithm for periodic [1] and sporadic [8], [9] task sets whose deadlines equal their periods.

We assume that tasks are independent and can be preempted at any time.

A task T_i is a program that repeatedly invokes jobs $T_{i,1}, T_{i,2}, \ldots$. Each task T_i is described using the 3-tuple (ϕ_i, p_i, e_i), where ϕ_i is T_i's offset, p_i is its period and e_i is its worst case execution time. Each job $T_{i,j}$ has a release time $a_{i,j}$, an execution requirement e_i and a relative deadline p_i — if $T_{i,j}$ arrives at time $a_{i,j}$ then it must be allowed to execute for e_i time units during the interval $[a_{i,j}, a_{i,j} + p_i[$. At any time $t \in [a_{i,j}, a_{i,j} + p_i[$, we say T_i's deadline at time t is $d_{i,j} = a_{i,j} + p_i$. If T_i is a periodic task set, then it invokes its first job at time $t = \phi_i$ and all the remaining jobs are invoked exactly p_i time units apart — i.e., $a_{i,j} = (j-1)p_i$ for all j. If T_i is a sporadic task, then it invokes its first job at any time $t \geq \phi_i$ and the remaining jobs are invoked no less than p_i time units apart — i.e., $a_{i,1} \geq 0$ and $a_{i,j} \geq a_{i,j-1} + p_i$ for all $j > 1$. A task set $\tau = \{T_1, T_2, \ldots, T_n\}$ denotes a set of n periodic or sporadic tasks. Throughout this paper, we will clearly state whether τ is assumed to be a periodic or a sporadic task set.

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

The LLREF scheduling algorithm partitions the time horizon into a sequence of intervals $[t_{j-1}, t_j[$, where $t_0 = 0$ and for each $j \geq 1$,

$$t_j = \min_{t > t_{j-1}} \{t = k \cdot p_i \mid T_i = (p_i, e_i) \in \tau \text{ and } k \in \mathbb{Z}\}.$$

For any time $t \geq 0$, we let $[t_{0_t}, t_{f_t}[$ denote the interval that contains t. At each time t, 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_{f_t}. 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). This plane is called the TL-plane. A task's *local utilization* is the proportion of time T_i must execute during the remainder of the current TL-plane, namely $r_{i,t} = \ell_{i,t}/(t_{f_t} - t)$. A task set's total utilization R_t at time t is defined to be the sum of each task's local utilization, viz., $R_t = \sum_{i=1}^{n} r_{i,t}$.

For our discussion, we need to be able to distinguish which tasks are *active* at any time t. We say a task T_i is active at time t if $\ell_{i,t} > 0$. We let $Active(t)$ and $NA(t)$ be the set of active and non-active tasks, respectively.

	LLREF	LRE-TL
Running time		
Initialize TL-plane	$O(n)$	$O(n)$
A events (per event)	NA	$O(\log n)$
B&C events (per TL-plane)	$O(n^2)$	$O(n \log n)$
Other overhead		
Max preemptions (per event)	$O(m)$	$O(1)$
Max migrations (per TL-plane)	$O(mn)$	$O(m)$

Table 1. Comparison of LLREF and LRE-TL.

We consider the problem of scheduling periodic and sporadic tasks on identical and uniform multiprocessors. Throughout this paper, we let m denote the number of processors. 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. We denote a uniform multiprocessor $\pi = [s_1, s_2, \ldots, s_m]$, where $s_i \geq s_{i+1}$ for $1 \leq i < m$. If processor s_i executes for t units of time, then it performs $s_i \times t$ units of work[1]. Below we discuss the implementation of LLREF and LRE-TL on identical multiprocessors first and then show how these algorithms can be extended for implementation on uniform multiprocessors.

3. A Brief Overview of LLREF

In order to understand the LRE-TL scheduling algorithm presented in this paper, we must first describe the LLREF algorithm presented in [7], where the authors also proved that LLREF is optimal for scheduling periodic task sets on identical multiprocessors. A task set τ can be scheduled to meet all deadlines on m identical processors if the following two conditions hold [6]

$$U(\tau) \leq m, \text{ and } u_{max}(\tau) \leq 1, \qquad (1)$$

where $u_{max}(\tau) = \max_{1 \leq i \leq n}\{u_i\}$ is the maximum utilization of all tasks in τ.

Given a task set τ satisfying the conditions above, LLREF schedules the tasks so that the *local* utilization continues to satisfy the stated conditions. 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

1. In a mild abuse of notation, we let s_i denote both the i^{th} processor and its speed.

deadlines within any TL-plane. Every task has a local execution requirement $\ell_{i,t}$, which denotes the amount of time the task must execute during the interval $[t, t_{f_t}[$. At the beginning of each TL-plane, $r_{i,t_{0_t}}$ is set to u_i for each task T_i (i.e, $\ell_{i,t_{0_t}} = u_i(t_{f_t} - t_{0_t})$).

LLREF makes scheduling decisions with the aim of achieving the following two goals.

1) No processor idles while a job is waiting, and
2) No task's local utilization ever exceeds 1.

Whenever LLREF makes a scheduling decision, it selects the m tasks with the largest local remaining execution and executes them until the next scheduling event. If fewer than m tasks have positive remaining execution, then LLREF executes all tasks T_i with $\ell_{i,t} > 0$ for $\ell_{i,t}$ time units. We use the following definitions to describe the tasks LLREF selects to execute at a given time t.

Definition 1. *At time t let $T_{(1)_t}, T_{(2)_t}, \ldots, T_{(n)_t}$ denote the tasks of τ sorted in weakly decreasing order according to local remaining execution. Thus,*

$$\ell_{(1)_t,t} \geq \ell_{(2)_t,t} \geq \ldots \geq \ell_{(n)_t,t}.$$

Let x_t denote the maximum number of tasks that can execute simultaneously at time t. Because, no more than m tasks can execute at one time, $x_t \leq m$. If fewer than m tasks have remaining, work, then x_t is the number of tasks with remaining work, viz.,

$$x_t = \min\{m, |Active(t)|\}. \tag{2}$$

When a scheduling event occurs at time t, LLREF executes tasks $T_{(i)_t}$ for $1 \leq i \leq x_t$. These tasks execute without interruption until one of two events occurs, namely a bottom (B) event or a critical (C) event. Below, we discuss these two events in more detail assuming tasks $T_{(i)_s}$ for $1 \leq i \leq x_s$ are scheduled at time s and the B or C event occurs at time t.

B (bottom) events: These events occur when a task hits the "bottom" of the TL-plane (i.e., when a task depletes its local remaining execution). Clearly, this can only happen to one of the x_s tasks that were scheduled to execute at time s. Using our sorting notation, task $T_{(x_s)_s}$ is the executing task with the least remaining execution requirement that triggers a B event at time t. If tasks are not rescheduled at this point, then the processor executing task $T_{(x_s)_s}$ will become idle, which could cause the first stated goal to be violated.

C (critical) events: These events occur when some task's local utilization becomes 1. Clearly, a task's local utilization decreases if it is executing. Hence, a C event is caused by the non-executing task with the largest remaining local execution, namely $T_{(m+1)_s}$. If this task is not allowed to execute immediately, its local utilization will exceed 1. This not only violates the second goal, it also means that $T_{(m+1)_s}$ will miss its deadline at time t_{f_t}.

Because each of these events cause one of LLREF's goals to be violated, they trigger a scheduling event. When a scheduling event occurs, LLREF determines which tasks are in $T_{(i)_t}$ for $1 \leq i \leq x_t$ and schedules these tasks to execute.

Two shortcomings of LLREF have been noted. First, it incurs fairly high overhead. Second, it is only optimal for periodic tasks with deadlines equal to periods. Below, we discuss how to address these two shortcomings.

4. LRE-TL: A Modification of LLREF

We now present our algorithm, which we call LRE-TL (local remaining execution-TL). The name LLREF does not accurately describe our scheduling algorithm's behavior because we no longer select to execute the task with the largest local remaining execution first. Below, we first present an analytical result that proves our proposed algorithm continues to be optimal. We then describe how the LRE-TL algorithm makes scheduling decisions and how to handle sporadic arrivals. Finally, we present the algorithm in detail and discuss its running time.

As noted above, LLREF has fairly high running time. When a scheduling event occurs at time t, LLREF must determine which tasks to execute (i.e., $T_{(i)_t}$ for $1 \leq i \leq x_t$). In addition, LLREF must determine when the next event will occur (i.e., it must find $\ell_{(x_t)_t,t}$ and $\ell_{(m+1)_t,t}$). This means the tasks must be at least partially sorted.

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. Say tasks are scheduled at time s and a B or C event occurs at time t. If we consider the tasks that executed during $[s, t[$ and the ones that did not execute during $[s, t[$ separately, it is easy to see that these tasks will maintain the same relative order at time t — i.e.,

$$\begin{array}{ll} \ell_{(i)_s,t} \geq \ell_{(j)_s,t} & \text{if } 1 \leq i \leq j \leq m \text{ and} \\ \ell_{(i)_s,t} \geq \ell_{(j)_s,t} & \text{if } m < i \leq j \leq n. \end{array}$$

Hence, the proper sorting order for the tasks at time t can be found by merging $\{T_{(i)_t} \mid 1 \leq i \leq m\}$ and $\{T_{(i)_t} \mid m+1 \leq i \leq n\}$. While this observation does reduce the running time, maintaining a sorted list is the most expensive portion of each scheduling event. This leads us to question whether this step is actually

necessary. How important is it that LLREF select the m tasks with *largest* remaining execution? Will any m tasks do, provided they have *non-zero* remaining execution? Below, we show that the answer to this question is "*yes*".

Recall the following result presented by Hong and Leung [10], [11]

Theorem 1 ([10], [11])**.** *No optimal online scheduler can exist for a set of jobs with two or more distinct deadlines for any m-processor identical multiprocessor, where $m > 1$.*

The theorem does not claim that no optimal algorithm exists if all deadlines are equal. In fact, Hong and Leung [10], [11] present an optimal algorithm when the jobs all have the same deadline. LLREF deliberately divides each job into subjobs so that (i) the work done by each subjob is proportional to the duration of the TL-plane, and (ii) at all times *all of the subjobs have the same deadline.* The key to our ability to both reduce LLREF's running time and define LLREF for sporadic tasks is the recognition that optimal multiprocessor algorithms do exist when deadlines are all equal.

Below, we make the simple observation that if x_t tasks execute at all times t within a TL-plane, the total utilization R_t decreases as time progresses.

Theorem 2. *Let τ be any task set executing on m identical processors. Let the timeline be divided into TL-planes and tasks be assigned local execution proportional to their utilization as in LLREF. Let s be any time such that $R_s \leq m$ and $r_{i,s} \leq 1$ for all tasks T_i. Let X be any x_s tasks that have positive local remaining execution at time s. Assuming the tasks in X are scheduled to execute at time s, let t_e be the time at which the next B or C event will occur. Then for any Δ such that $0 \leq \Delta \leq t_e - s$,*

$$R_{t_e} \leq R_{s+\Delta} \leq R_s. \tag{3}$$

Moreover, if $R_s < m$ then $R_{t_e} < R_{s+\Delta} < R_s$.

In other words, if $U(\tau) \leq m$ and x_t tasks execute at all times t, then the total local utilization never increases value within any TL-plane, and it constantly decreases if $U(\tau) < m$.

Proof: Because the total work done during the interval $[s, s+\Delta]$ is equal to $x_s \times \Delta$, we know that $\sum_{i=1}^{n} \ell_{i,s+\Delta} = \sum_{i=1}^{n} \ell_{i,s} - x_s \times \Delta$. Therefore, we can determine $R_{s+\Delta}$ as follows.

$$\begin{aligned} R_{s+\Delta} &= \sum_{i=1}^{n} \frac{\ell_{i,s}}{t_f - s - \Delta} - \frac{x_s \times \Delta}{t_f - s - \Delta} \\ &= \sum_{i=1}^{n} \frac{\ell_{i,s}}{t_f - s} \times \frac{t_f - s}{t_f - s - \Delta} - \frac{x_s \times \Delta}{t_f - s - \Delta} \end{aligned}$$

Note that

$$\frac{t_f - s}{t_f - s - \Delta} = 1 + \frac{\Delta}{t_f - s - \Delta}$$

Therefore,

$$\begin{aligned} R_{s+\Delta} &= R_s + \frac{\Delta(R_s - x_s)}{t_f - s - \Delta} \\ &\leq R_s \end{aligned}$$

The last step follows both when $|Active(s)| \geq m$ and when $|Active(s)| < m$. In the first case, $R_s - x_s = R_s - m \leq 0$, because $R_s \leq m$. In the second case, $x_s = |Active(s)|$. Because $r_{i,s} \leq 1$ for all T_i, we can conclude that $R_s - x_s \leq 0$ in this case as well. □

With this theorem in mind, we make the following observations, which allow us to modify LLREF as described below..

Observation 1 The total local utilization decreases *regardless of which tasks execute* provided that x_t processors execute at all times t. **Observation 2** The total local utilization at the beginning of a TL-plane $[t_0, t_f[$ is equal to $\sum_{T_i \in Active(t_0)} u_i$. Hence, if a sporadic task T_i generates a job at time $t \in [t_0, t_f[$, the local utilization just prior to t will be at most $m - u_i$.

4.1. A Simplifying Observation

As noted above, the most expensive portion of processing a scheduling event is re-sorting the tasks. Theorem 2 above allows us to remove this step from the algorithm. We propose to maintain two heaps – one heap for each type of event. 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. H_B is a min heap whose key is $(t + \ell_{i,t})$. Heap H_C is the set of active tasks that are not executing. Task $T_i \in H_C$ triggers a C event if it does not execute before time $(t_{f_t} - \ell_{i,t})$, wheret_{f_t} denotes the end of the current TL-plane. H_C is a min heap whose key is $(t_{f_t} - \ell_{i,t})$.

Note that a task T_i's key does not change as long as T_i remains in the same heap. The value of t_{f_t} remains constant within any TL-plane. If $T_i \in H_C$, then $\ell_{i,t}$ is not changing over time, so $(t_{f_t} - \ell_{i,t})$ remains constant.

Also, if $T_i \in H_B$, then $\ell_{i,t}$ decreases as t increases, so the $(t + \ell_{i,t})$ remains constant.

If a task T_i switches from one heap to the other at time t, then its new key is set to $(t_{f_t} - T_i.\text{key}+t)$, where T_i.key was its key prior to switching heaps. This observation allows us to maintain proper execution without having to update ℓ at each B or C event.

The modified algorithm will only preempt a task if it is absolutely necessary to do so. If a B event occurs, any task that was executing just prior to the B event will continue to execute (on the same processor) after the B event is handled. If a C event occurs, the task that triggered the C event must execute immediately. Therefore it will preempt one of the executing tasks and execute on that task's processor. Thus, a scheduling event at time t causes at most $\nu(t)$ preemptions, where $\nu(t)$ is the number of tasks whose total local utilization increase to 1 at time t.

The algorithm LRE-TL is described in more detail in Subsection 4.3. First, though, we discuss how to handle sporadic task arrivals in the middle of a TL-plane.

4.2. Scheduling Sporadic Tasks

In this subsection, we show how to handle the arrival of a job invoked by a sporadic task and we demonstrate that this method will not allow any deadline misses provided τ is a feasible task set. Assume a task T_s invokes a job at time t_s in the middle of a TL-plane. By Theorem 2 above, we know that the total local utilization is at most $(m - u_s)$ just prior to T_s's arrival. Therefore, we can set T_s's local execution to be proportional to its utilization, just as we would at the beginning of a TL-plane. Specifically,

$$\ell_{s,t_s} = u_s \cdot (t_{f_{t_s}} - t_s). \tag{4}$$

The theorem below shows that adding T_s to the set of active tasks at time t_s will not cause any other tasks to miss their deadlines. If, in addition, T_s does not have a deadline within the current TL-plane, it will also be guaranteed to meet its deadline.

Theorem 3. *Let τ be a sporadic task set such that $U(\tau) \leq m$ and $u_{max}(\tau) \leq 1$. Assume τ is scheduled using LRE-TL as described above. Let $[t_0, t_f[$ be a TL-plane for the LRE-TL schedule of τ. Assume task T_s is not active at time t_0 and becomes active at some time $t_s \in [t_0, t_f[$. If the algorithm sets ℓ_{s,t_s} according to Equation 4, then T_s will not cause any tasks to miss their deadlines. If, in addition, $t_s + p_s \geq t_f$, then τ can be scheduled to ensure T_s will meet its deadline at time $(t_s + p_s)$.*

Proof: We first argue that T_s will not cause other tasks to miss their deadlines and then demonstrate how to ensure that T_s will not miss its deadline.

Because T_s is not active at time t_0, we know $R_{t_0} \leq m - u_s$. By Theorem 2, just prior to t_s we know that $R_{t_s-\epsilon} \leq R_{t_0}$. Therefore, upon setting the value of ℓ_{s,t_s} to $u_s \cdot (t_f - t_s)$, making $r_{s,t_s} = u_s$, we know that $R_{t_s} \leq m$. Hence, once ℓ_{s,t_s} is added to the total local remaining execution it will still be possible to meet all local execution requirements by time t_f. As with all tasks, the amount of work T_s will have completed at time t_f will be the product of T_s's utilization and the total amount of time since it arrived. Hence, in subsequent TL-planes, T_s can have its local utilization set to $r_{s,t_0} = u_s$. Because $U(\tau) \leq m$, the total utilization of all TL-planes continues to be at most m. Hence, T_s will not cause any other tasks to miss their deadlines.

It remains to demonstrate that we can schedule τ to ensure that T_s will not miss its deadline. Because we assume $t_f \leq t_s + p_s$, we know that T_s will not have a deadline before time t_f. If we ensure that there is a TL-plane that ends at time $(t_s + p_s)$, then T_s will meet its deadline. Algorithm LRE-TL has control over the TL-planes and can ensure that this will incur. Therefore, T_s will meet its deadline at time $(t_s + p_s)$. □

The correctness of LLREF (and of LRE-TL) hinges on having (i) all tasks complete their local execution by the end of every TL-plane, and (ii) every task's deadline coincide 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, we determine the duration of the TL-plane by finding the earliest upcoming deadline, d_{next}, and setting t_f equal to d_{next}. Doing this will ensure that all jobs of active tasks will have deadlines that correspond with the end of some TL-plane. In addition, we must ensure that non-active sporadic tasks will not have a deadline within the TL-plane. If we ensure that no TL-plane has a duration longer than p_{min}, the minimum task period of all tasks in τ, then for any job generated by a sporadic task there must be an TL-plane break between the job's arrival and deadline.

Hence, we take the following two steps in order to handle sporadic tasks.

TL-plane Boundaries Instead of setting TL-plane boundaries to be $k \cdot p_i$, where $k \geq 0$ and $1 \leq i \leq n$, we base TL-plane boundaries on the deadlines of active jobs. Let t_0 be the beginning of some TL-plane. We

Algorithm 1 LRE-TL

1: **if** $t_{cur} = t_f$ **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: $t_{next} \leftarrow H_B$.min-key()
10: **if** H_C.size()> 0 **then**
11: $t_{next} \leftarrow \min\{t_{next}, H_C\text{.min-key()}\}$
12: **else**
13: $t_{next} \leftarrow t_f$
14: let each processor execute its designated task
15: sleep until time t_{next}

Algorithm 2 TL-Plane-Initialize

Require: $Active$ contains the set of all tasks that are currently active. H_B and H_C are both empty. Task T_{min} has the shortest period of all tasks (whether active and non-active).

1: **for all** tasks T_i that arrived at time t_{cur} **do**
2: **if** H_D.find-key($t_{cur} + p_i$) = NULL **then**
3: H_D.insert($t_{cur} + p_i$)
4: $t_f \leftarrow t_{cur} + p_{min}$
5: **if** H_D.min-key() $\leq t_f$ **then**
6: $t_f \leftarrow H_D$.extract-min()
7: $z \leftarrow 1$
8: **for all** $T_i \in Active$ **do**
9: $\ell \leftarrow u_i(t_f - t_{cur})$
10: **if** $z \leq m$ **then**
11: T_i.key $\leftarrow t_{cur} + \ell$
12: T_i.proc-id $\leftarrow z$
13: z.task-id $\leftarrow T_i$
14: H_B.insert(T_i)
15: $z \leftarrow z + 1$
16: **else**
17: T_i.key $\leftarrow t_f - \ell$
18: H_C.insert(T_i)
19: **for all** processors z' s.t. $m \geq z' > z$ **do**
20: z'.task-id $\leftarrow$ NULL

determine t_f, the end of the TL-plane as follows. Let d_{next} be the earliest upcoming deadline and let p_{min} be the shortest task period ($p_{min} = \min\{p_i \mid 1 \leq i \leq n\}$). Then $t_f = \min\{d_{next}, t_0 + p_{min}\}$. This ensures that all jobs' deadlines coincide with the end of some TL-plane. We add one new heap H_D, which contains all current deadlines, to implement this modification efficiently.

The A Event We introduce a new event, namely the A (arrival) event. When a sporadic task T_s invokes a new job it triggers an A event. During an A event, ℓ_{s,t_s} is set to $u_s(t_{f_{t_s}} - t_s)$ and T_s is added to one of the heaps (H_B or H_C). Under most circumstances, T_s will be added to H_C, the heap containing the non-executing tasks, and execution will resume without preempting any tasks. However, if $u_s = 1$, then clearly T_s must preempt an executing task. Additionally, T_s will be scheduled to execute immediately without preempting any other tasks if $x_{t_s} < m$.

We have explored LRE-TL and justified that none of the modifications to LLREF sacrifice optimality. We now describe the algorithm in more detail.

4.3. Algorithm LRE-TL

The algorithm LRE-TL is comprised of four procedures. The main algorithm 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. 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, and puts the task in one of the heaps (H_B or H_C). The B and C event handler maintains the correctness of H_B and H_C.

Throughout this section, we discuss executing *tasks* rather than executing *jobs*. Recall deadlines are equal to periods and every deadline coincides with the end of a TL-plane. Therefore, given any TL-plane $[t_0, t_f[$ and any task T_i, at most one job of T_i overlaps with $[t_0, t_f[$. Because all scheduling decisions are made within a TL-plane, 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.

At all times, each active task T_i will be in exactly one task heap (H_B or H_C). Throughout this section, 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.

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 H. H.insert(I) inserts item I into the heap. H.find-key(k) returns

Algorithm 3 LRE-TL-A-Event

Require: The sporadic task T_s that invokes a job to trigger this algorithm is not in $Active$.

```
1:  ℓ ← u_s(t_f − 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_f − 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_f − 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 + p_s) = NULL then
21:     H_D.insert(t_cur + p_s)
```

a pointer to the object whose key equal 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 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_f (lines 1 through 6), which is set to the earliest upcoming deadline, but is never larger than $(t_{cur} + p_{min})$, where $p_{min} = \min\{p_i \mid 1 \leq i \leq n\}$. Once t_f is identified, the local execution values of active tasks are initialized accordingly (lines 8 through 18). 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 active tasks, the idle processors' task id's are set to NULL (lines 19 through 20).

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 all m processors are busy, 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 some executing task T_b is moved from H_B to H_C if $u_s = 1$ (lines 12 through 19). Before returning, T_s's deadline is inserted into H_D (lines 20 through 21).

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) either all processors are busy or all 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. Run Time Analysis

The running time of LRE-TL depends on which routine it calls. Below we establish the running time for each event handler and compare LRE-TL's running time to LLREF's running time. This comparison is summarized in Table 1.

The TL-plane initializer has a loop that iterates $O(n)$ times. During these iterations, the two heaps are populated, which takes $O(\log(n-m) + \log m)$ time. This gives an overall running time of $O(n + \log(n - m) + \log m)$. Assuming $m \leq n$, the initializer takes $O(n)$ 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, this running time cannot be compared to that of LLREF.

The running time of the B and C event handler depends on the number of active 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. Hence,

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_f.

```
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_f − 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_f − T_b.key+t_cur
17:         T_c.key← t_f − 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)
```

the total running time within any TL-plane is

$$\begin{aligned}\sum_{\alpha=1}^{m} \log \alpha \quad &+ \quad \sum_{\alpha=m+1}^{n} (\log m + \log(\alpha - m)) \\ &= O(n \log m + (n-m)\log(n-m)) \\ &= O(n \log n).\end{aligned}$$

By contrast, LLREF needs to update ℓ_i for the executing tasks T_i and establish the new sort order if any task are forced to wait. Updating ℓ takes $O(m)$ time if all processors are busy and $O(\alpha)$ time otherwise. As established earlier, the tasks can be re-sorted through a simple merge, which takes $O(\alpha)$ time. This gives a total running time of $\sum_{\alpha=1}^{m} \alpha + \sum_{\alpha=m+1}^{n} (\alpha + m) = O(n^2)$.

There is one benefit of LRE-TL that is not captured in the discussion of running time – namely, the reduction overhead due to fewer preemptions and migrations. Because LLREF sorts the tasks upon every B or C event, a single event could cause m tasks to be preempted. Upon resuming execution, each of these tasks might end up on a different processor. Thus, the maximum number of preemptions and migrations within each TL-plane is $O(mn)$. By contrast, LRE-TL preempts only when absolutely necessary – i.e., only when a C event occurs. Because utilization decreases over time, the number of C events within a TL-plane is at most $(m-1)$. Hence the number of preemptions and migrations is $O(m)$.

Using the above analysis, we can determine the maximum scheduling overhead per TL-plane. We can build this overhead the schedulability test given in Equation 1. One simple way of doing this would be to evenly distribute this overhead among the tasks. Let v denote the maximum total scheduling overhead per TL-plane and κ_i denote the maximum number of time slices required to schedule any job of task T_i. Then we could distribute this overhead among the tasks and modify the task utilization accordingly. For each T_i, the modified utilization would be $u'_i = (e_i + \kappa_i(v/n))/p_i$, or $u'_i = u_i + (\kappa_i/p_i)(v/n)$. Then τ is LRE-TL schedulable on m processors if $u'_{max} \leq 1$ and $U'(\tau) \leq m$, where u'_{max} and $U'(\tau)$ are the maximum and total values of u'_i over all tasks T_i. This overhead accounting could be improved by allocating more of the overhead to the task T_i with the minimum ratio of κ_i to p_i provided T_i's modified utilization does not exceed 1.

5. Example

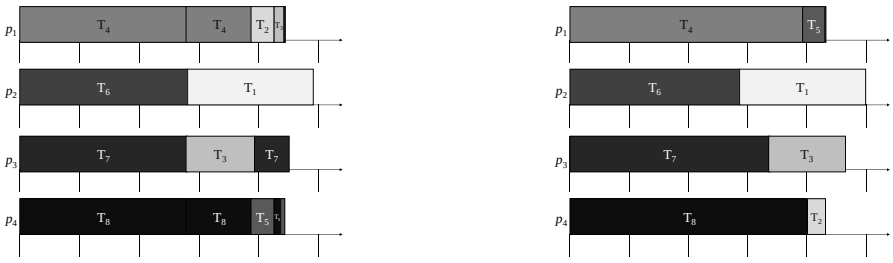
In this section we present the LLREF and LRE-TL schedule of the task set shown in Table 2, which was used in [7]. We illustrate the schedule on 4 processors for the first TL plane, $[0, 5[$.

	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

Table 2. Demonstration task set.

The two schedules are shown in Figure 1. Each timeline corresponds to one of the four processors. Figure 1(a) contains the LLREF schedule and Figure 1(b) contains the LRE-TL schedule.

As expected, LRE-TL has fewer preemptions and migrations. Even for this small example, the difference is quite stark. LLREF triggers 5 preemptions, 2 of



(a) LLREF (b) LRE-TL

Figure 1. Schedule comparison.

which result in a task migration, whereas LRE-TL triggers only 1 preemption and migration[2]. (Some preemptions and migrations are not visible in the figures because the remaining execution is so small when the preemption occurs.)

There are several points in the LLREF 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. By contrast, 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 becomes critical. Note that because $\ell_{(4)_0,0} + \ell_{(5)_0,0} > 5$, there must be at least one preemption regardless of what scheduling algorithm is used.

6. LRE-TL for Uniform Multiprocessors

Recently [12] LLREF was extended so that it can now be scheduled on uniform multiprocessors as well as identical multiprocessors. A periodic task set τ can be successfully scheduled on a uniform multiprocessor π provided the following conditions are satisfied [13]

$$\sum_{i=1}^{k} u_i \leq \sum_{i=1}^{k} s_i \text{ for } 1 \leq k < m, \text{ and} \quad (5)$$

$$U(\tau) \leq \sum_{i=1}^{m} s_i. \quad (6)$$

Chen and Hsueh [12] presented an extension of LLREF using these two conditions. If τ satisfies the above conditions for uniform multiprocessor π, then their extension to LLREF that ensures the above two conditions apply for the local utilization at all times. Their algorithm requires that at the beginning of the TL plane the m tasks with the largest local remaining execution $T_{(1)_0}, T_{(2)_0}, \ldots, T_{(m)_0}$ are scheduled to execute with task $T_{(i)_0}$ being assigned to processor s_i. As with identical multiprocessors, their algorithm triggers a B event at time t_b when some task T_b completes its execution time — i.e., $\ell_{b,t_b} = 0$. C events, however, are handled differently. A task T_c becomes critical at time t_c if T_c's utilization is equal to the speed of some processor s_i — i.e., $r_{c,t_c} = s_i$. When such an event occurs, T_c is assigned to processor s_i and s_i is removed from further consideration.

6.1. Sporadic Tasks on Uniform Multiprocessors

We now show that LRE-TL can optimally schedule sporadic tasks in addition to periodic tasks on uniform multiprocessors. We schedule sporadic tasks in exactly the same manner as they are handled for identical multiprocessors with one important exception. If a set of k tasks has total utilization equal to the sum of the k fastest processors, then these k tasks must execute on the k fastest processors *for the remainder of the TL-plane* in order to guarantee they meet their deadlines. Hence, if a sporadic task T_s is among these k tasks, it must be scheduled on one of the k fastest processors as soon as it arrives. With this in mind, we define the following terms.

Definition 2. *Let τ be a task set that is feasible on some uniform multiprocessor π. For $1 \leq k < m$, let $Crit_k(\tau, \pi)$ be* TRUE *if the k highest-utilization tasks in τ have total utilization greater than the total speed of the $(k+1)$ fastest processors.*

$$Crit_k(\tau, \pi) = \left(\sum_{i=1}^{k} s_i \geq \sum_{i=1}^{k} u_i > \sum_{i=1}^{k+1} s_i \right).$$

2. 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.

If $Crit_k(\tau, \pi)$ is TRUE *for some k then tasks T_1 through T_k will need to dominate the k fastest processors.*

Assume that the tasks are indexed in decreasing order by utilization – i.e., task T_j has the j^{th} largest utilization (ties can be broken arbitrarily). Let $MinProc(T_j)$ be the minimum processor speed that T_j can safely use – i.e., if $j < m$ is the smallest $k \geq j$ for which$Crit_k(\tau, \pi)$ is TRUE*. Otherwise,$MinProc(T_j) = m$.*

We use $MinProc(T_s)$ to determine whether or not the sporadic task T_s must preempt some executing task when it invokes a new job in the middle of a TL-plane. Specifically, if $MinProc(T_s) = m$, then T_s can be handled in the same manner as described in Algorithm 3. If, however, $MinProc(T_s) < m$, then T_s must execute on one of the $MinProc(T_s)$ fastest processors when it invokes a job. By definition, exactly k tasks will have $MinProc$ values less than or equal to k. Therefore, there will be some task T_j such that $MinProc(T_j) > MinProc(T_s)$ and T_j is executing on one of the $MinProc(T_s)$ processors. The arriving sporadic task T_s can preempt any such task T_j. Provided this step is taken, LRE-TL will ensure no tasks miss their deadlines on uniform multiprocessors.

7. Conclusion

This paper presents a simple observation which has far reaching impact on the efficiency of the LLREF scheduling algorithm: Local task utilization constantly decreases within a TL-plane provided no processor idles while tasks are waiting to execute. Using this observation, we have significantly reduced the overhead of the LLREF algorithm – both by shortening the running time and by reducing the number of preemptions and migrations. We call our new algorithm LRE-TL and demonstrate that it can optimally schedule sporadic task sets on both identical and uniform multiprocessors.

We briefly discussed how scheduling overhead can be handled in determining LRE-TL schedulability. In the future, we plan to explore this approach more fully.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] S. Davari and S. K. Dhall, "On a real-time task allocation problem," in *Proceedings of the 19th Hawaii International Conference on System Science*, Honolulu, January 1985.

[3] T. Baker, "Multiprocessor edf and deadline monotonic schedulability analysis," in *24th Real-Time Systems Symposium*, 2003.

[4] ——, "An analysis of edf schedulability on a multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 8, pp. 760 – 768, 2005.

[5] C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 4–6 May 1997, pp. 140–149.

[6] S. K. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, June 1996.

[7] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Proceedings the 27^{th} IEEE Real-Time System Symposium (RTSS)*, Los Alamitos, CA, 2006, pp. 101 – 110.

[8] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[9] M. Dertouzos, "Control robotics : the procedural control of physical processors," in *Proceedings of the IFIP Congress*, 1974, pp. 807–813.

[10] K. S. Hong and J. Y.-T. Leung, "On-line scheduling of real-time tasks," in *Proceedings of the Real-Time Systems Symposium*. Huntsville, Alabama: IEEE, December 1988, pp. 244–250.

[11] ——, "On-line scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 41, pp. 1326–1331, 1992.

[12] S.-Y. Chen and C.-W. Hsueh, "Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors," in *Proceedings of the 2008 Real-Time Systems Symposium*, pp. 147–156,.

[13] S. Funk, J. Goossens, and S. K. Baruah, "On-line scheduling on uniform multiprocessors," in *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 2001, pp. 183–192.