T–L plane-based real-time scheduling for homogeneous multiprocessors

Hyeonjoong Cho a,∗, Binoy Ravindran b, E. Douglas Jensen c

a Department of Computer and Information Science, Korea University, South Korea
b ECE Department, Virginia Tech., Blacksburg, VA 24061, USA
c The MITRE Corporation Bedford, MA 01730, USA

ARTICLE INFO

Article history:
Received 5 August 2008
Received in revised form 10 November 2009
Accepted 16 December 2009
Available online xxxx

Keywords:
Real-time scheduling
Optimality
Multiprocessor systems

ABSTRACT

We consider optimal real-time scheduling of periodic tasks on multiprocessors—i.e., satisfying all task deadlines, when the total utilization demand does not exceed the utilization capacity of the processors. We introduce a novel abstraction for reasoning about task execution behavior on multiprocessors, called T–L plane and present T–L plane-based real-time scheduling algorithms. We show that scheduling for multiprocessors can be viewed as scheduling on repeatedly occurring T–L planes, and feasibility scheduling on a single T–L plane results in an optimal schedule. Within a single T–L plane, we analytically show a sufficient condition to provide a feasible schedule. Based on these, we provide two examples of T–L plane-based real-time scheduling algorithms, including non-work-conserving and work-conserving approaches. Further, we establish that the algorithms have bounded overhead. Our simulation results validate our analysis of the algorithm overhead. In addition, we experimentally show that our approaches have a reduced number of task migrations among processors when compared with a previous algorithm.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Multiprocessor architectures (e.g., Symmetric Multi-Processors or SMPs, Single Chip Heterogeneous Multiprocessors or SCHMs) are becoming more attractive for embedded systems, primarily because major processor manufacturers (Intel and AMD) are rapidly decreasing the prices and increasing the cost/performance. Responding to this trend, real-time operating system (RTOS) vendors are increasingly providing multiprocessor platform support. But, this exposes the critical need for real-time scheduling for multiprocessors—a comparatively undeveloped area of real-time scheduling which has recently received significant research attention, but is not yet well supported by the RTOS products. Consequently, the impact of cost-effective multiprocessor platforms remains nascent.

Liu [9] first addressed the multiple-resource scheduling problem that considers allocating several identical resources (e.g., multiple processors) to a number of periodic tasks, where a task is characterized by two parameters, execution time and period. A valid schedule must satisfy two constraints: (1) at any instant, at most one task can be executed on any single processor; and (2) no single task can be executed on more than one processor at the same time instant. Under these constraints, a feasible real-time schedule allocates the exact time units of the resource that each task requires for its execution to the task during its period.

One unique aspect of multiprocessor scheduling is the degree of run-time migration that is allowed for job instances of a task across processors (at scheduling events). Example migration models include: (1) full migration, where jobs are allowed to arbitrarily migrate among processors during their execution. This usually implies a global scheduling strategy, where a single shared scheduling queue is maintained for all processors, and a system-wide scheduling decision is made by a single (global) scheduling algorithm; (2) no migration, where tasks are statically (off-line) partitioned and allocated to processors. At run-time, job instances of tasks are scheduled on their respective processors by processors’ local scheduling algorithm, like single processor scheduling; and (3) restricted migration, where some form of migration is allowed—e.g., at job boundaries.

Carpenter et al. [3] have catalogued multiprocessor real-time scheduling algorithms considering the degree of job migration and the complexity of priority mechanisms employed. The latter includes classes such as (1) static, where task priorities remain fixed, e.g., rate-monotonic (RM); (2) dynamic but fixed within a job, where job priorities are fixed, e.g., earliest-deadline-first (EDF)1 and (3) fully dynamic, where job priorities are dynamic.

* Corresponding author.
E-mail addresses: raycho@korea.ac.kr (H. Cho), binoy@vt.edu (B. Ravindran), jensen@mitre.org (E.D. Jensen).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.

Please cite this article in press as: H. Cho, et al., T–L plane-based real-time scheduling for homogeneous multiprocessors, J. Parallel Distrib. Comput. (2010),
The *Proportionate fair* (Pfair) algorithms [2] that allow full migration and fully dynamic priorities have been shown to be theoretically optimal—i.e., they achieve a schedulable utilization bound (below which all tasks meet their deadlines) that equals the total capacity of all processors. Fairness allows all tasks to receive a share of the processor time and simultaneously make progress. However, Pfair algorithms incur significant run-time overhead due to their quantum-based scheduling approach [6,11]—under Pfair, tasks can be decomposed into several small uniform segments, which are then scheduled, causing frequent scheduling and migration.

Zhu et al. proposed *boundary fair* (BF) scheduling, which makes scheduling decisions only at period boundaries to reduce the number of scheduling points [12]. It is not as fair as Pfair (fair at any time quantum), but fair enough (only at period boundaries) to get a feasible schedule. Especially when the number of tasks is small (less than 100 in their experiments), they showed that the overhead of BF is less than that of PD (an efficient Pfair algorithm). Note that as the number of tasks increases, the frequency of boundaries also increases and consequently, the number of scheduling points of BF becomes similar to that of PD.

In this paper, we focus on the multiple-resource scheduling problem. We introduce an abstraction for reasoning about the execution behavior of a class of periodic tasks on multiprocessors, *time and local remaining execution-time plane* (abbreviated as the T–L plane). The T–L plane makes it possible to envision the entire scheduling activity over time as scheduling in repeated T–L planes of various sizes, so that feasibly scheduling on a single T–L plane results in an optimal schedule for all T–L planes across time.

Moreover, the T–L plane provides a visual model of task execution behavior on multiprocessors, which allows insightful and analytical understanding. Based on this in-depth understanding from the visual model, we present the minimum but sufficient guidelines for designing feasible scheduling algorithms for our task model on multiprocessors. This provides flexibility in the sense that task priorities can be assigned as desired within the constraints of the sufficiency guidelines.

More concretely, T–L plane-based scheduling consists of two phases: the *local parameter decision phase* (LD-P) to establish a T–L plane over a period of time, and the *local scheduling phase* (LS-P) to locally schedule tasks within the established T–L plane. First focusing on LS-P, we analytically derive a sufficient condition to provide a feasible schedule within a single T–L plane. Based on this, we provide two examples of T–L plane-based real-time scheduling algorithms: one non-work-conserving and one work-conserving. The work-conserving approach is for reducing task response times, while the non-work-conserving approach smooths out the task completion flow. In addition, we experimentally show that our approach outperforms an existing algorithm (McNaughton’s [10], which Zhu et al. used in [12]) for local scheduling by reducing unnecessary migration.

Thus, the paper’s contributions include the T–L plane scheduling abstraction for optimal multiprocessor real-time scheduling of a class of periodic tasks. We also provide a set of sufficient conditions for feasible scheduling of that class. Finally, we provide two T–L plane-based real-time scheduling algorithms—one work-conserving and one non-work-conserving. These have bounded scheduling overhead and simultaneously, lower number of task migrations compared to the existing algorithm mentioned above.

The rest of the paper is organized as follows: In Section 2, we discuss the rationale behind the T–L plane. In Section 3, we analytically discuss the properties of T–L plane-based real-time scheduling. In Section 4, we establish two examples of T–L plane-based scheduling algorithms and establish the upper bound of the algorithms’ overhead. Several experimental results are discussed in Section 5. The paper concludes in Section 6.
Fig. 1 illustrates the fundamental idea behind the T–L plane. For a task \( T \) with \( a_i, c_i, \) and \( d_i \), the figure shows a 2-dimensional plane with time represented on the x-axis and the task's remaining execution time represented on the y-axis. If \( a_i \) is assumed as the origin, the dotted line from \((0, c_i)\) to \((d_i, 0)\) depicts the fluid schedule, the slope of which is \(-\frac{c_i}{d_i}\). Since the fluid schedule is ideal but practically impossible, the fairness of a scheduling algorithm depends on how much the algorithm approximates the fluid schedule path.

When \( T \) runs like in Fig. 1, for example, its execution can be represented as a broken line between \((0, c_i)\) and \((d_i, 0)\). Note that task execution is represented as a line whose slope is \(-1\) since x and y axes are in the same scale, and the non-execution over time is represented as a line whose slope is zero. It is clear that the Pfair algorithm can also be represented on the T–L plane as a broken line based on time quanta.

When \( N \) number of tasks are considered, their fluid schedule can be constructed as shown in Fig. 2. After constructing the fluid schedule of each task over time, two consecutive scheduling events are considered, \( t_1 \) and \( t_2 \), for example. Then, a horizontal line for each task is established from the fluid schedule at the following scheduling event \( t_2 \). The length of the horizontal line is \( t_2 - t_1 \), which is, not surprisingly, the same as those of all other tasks between \( t_1 \) and \( t_2 \). At the previous scheduling event \( t_1 \), a vertical line of the same length as the horizontal line is drawn for each task, and together with the horizontal line, this yields a right isosceles triangle between \( t_1 \) and \( t_2 \) for each task. Since the triangles between two consecutive scheduling events for each task are of same size, those \( N \) triangles can be overlapped with the fluid schedule of each task inside. The whole schedule over time can be represented as the sequence of these overlapped triangles, \( TL_1, \ldots, TL_k, \ldots \), where \( k \) is simply increasing over time. We call this triangle the T–L plane. The size of the kth T–L plane, \( TL_k \), may change over \( k \). The bottom side of each triangle represents time. The left vertical side of each triangle represents the task's remaining execution time, which we call the local remaining execution time.

Definition 1 (Local Remaining Execution Time). \( TL_k(t) \) denotes the \( k \)th task's local remaining execution time at time \( t \) on the \( k \)th T–L plane, which is supposed to be consumed before the end of the \( k \)th T–L plane.

\( T \)'s deadline may not coincide with the end of \( TL_k \), as more than one T–L plane often exists between a task's release and completion. Fluid schedules for tasks are constructed to be overlapped in each \( TL_k \) plane, which keeps their slopes the same.

The abstraction of T–L planes is significantly meaningful in scheduling our task model for multiprocessors, because T–L planes are repeated over time, and a feasible scheduling algorithm for a single T–L plane leads to feasible scheduling of all tasks on all repeated T–L planes.

2.3. Scheduling on T–L planes

Fig. 3 details a single T–L plane. (When \( k \) is omitted, it implicitly means the current \( k \)th T–L plane, \( TL_k \). We explicitly use the index \( k \) when necessary, for example, when comparing two adjacent T–L planes.)

The status of each task is represented as a token on the T–L plane. The token's location describes the current time as a value on the horizontal axis and the task's remaining execution time as a value on the vertical axis. The remaining execution time of a task here means the time that must be consumed before the finishing time of the T–L plane, \( t_f \), and not the task's deadline. Hence, we call it the \textit{local} remaining execution time.

As scheduling decisions are made over time, each task's token moves on the T–L plane. Although the ideal paths of tokens are shown as dotted lines in Fig. 3, the tokens are allowed to move only on two paths. (Therefore, tokens can deviate from the ideal paths.) When the task is selected and executed, the token moves diagonally down, as \( T \) moves. Otherwise, it moves horizontally, as \( T \) moves. If \( M \) processors are being scheduled and executing, at most \( M \) tokens can diagonally move together. The scheduling objective on the T–L plane is to make all tokens arrive at the bottom of the T–L plane before \( t_f \) i.e., to make the local remaining execution times of all tasks be consumed before \( t_f \). We call this successful arrival \textit{locally feasible}. If all tokens are locally feasible on each T–L plane, it is possible for them to be scheduled throughout all consecutive T–L planes over time, approximating all tasks' ideal paths.

For convenience, we define the \textit{local laxity} of a task \( T \).
**Definition 2 (Local Laxity).** The local laxity of the task $T_i$ at the current time $t_{\text{cur}}$ is denoted as $t_f - t_{\text{cur}} - l_i(t_{\text{cur}})$, where $1 \leq i \leq N$.

The diagonal of the T–L plane has an important meaning: when a token reaches that side, it indicates that the task does not have any local laxity. Thus, if it is not selected immediately, then it cannot satisfy the scheduling objective of local feasibility. We call the diagonal of the T–L plane the **no local laxity diagonal** (or NLLD). All the tokens are supposed to stay on or under the NLLD. We call the tokens both on NLLD and between NLLD and the bottom line of the T–L plane **active**. Active tokens have non-zero local remaining execution time. We call the tokens lying on or below the bottom line of the T–L plane **inactive**.

We observe that there are two time instants when the scheduling decision has to be made again on the T–L plane. One instant is when the local remaining execution time of a task is completely consumed, and it would be better for the system to run another task instead. When this occurs, the token reaches the horizontal line, as $T_k$ does in Fig. 3. We call this the **bottom reaching event** (or event-B). The other instant is when the local laxity of a task becomes zero so that the task must be selected immediately. When this occurs, the token reaches the NLLD, as $T_j$ does in Fig. 3. We call this the **ceiling reaching event** (or event-C). To distinguish these events from traditional scheduling events, we call events B and C **secondary events**.

To design a scheduling algorithm for a single T–L plane, we propose guidelines for **locally-feasible scheduling (GLS)**, which are sufficient to provide local feasibility. (The sufficiency of the GLS is proved in Section 3.)

1. At every scheduling event, as many active tokens as the number of processors allows (up to M) should be selected. (GLS.1)
2. At event-C, the token that invokes the event should be selected immediately. (GLS.2)
3. Since an active token changes to inactive when it reaches the bottom line of the T–L plane, (GLS.1) implies following (GLS.3).

- At event-B, the token that invokes the event should not be selected if there are still at least M active tokens.

The fact that obeying these GLS’s is sufficient for a scheduling policy to be able to provide local feasibility in a T–L plane allows significant flexibility in designing a scheduling algorithm. For example, a scheduling algorithm may assign task priorities for a particular application while ensuring local feasibility by following the GLS’s. The priority assignment also could be either work-conserving or non-work-conserving, specify processor affinity, etc. At time $t_f^k$ of TL, the time instant for the next task period event, the next task period next T–L plane TL$^{k+1}$ starts to be drawn and the designed scheduling algorithm remains valid. Thus, the scheduling algorithm is consistently applied for every event.

We consider local scheduling algorithms (LSA) designed following the GLS’s. In the next section, we analytically prove that an LSA provides local feasibility in a T–L plane.

**3. Properties of LSAs**

A fundamental property of an LSA is its local feasibility—i.e., an LSA can consume all task local remaining execution times before the T–L plane ends. In this section, we prove that an LSA guarantees local feasibility on the T–L plane. We first suppose the case that $N > M$. The case that $N \leq M$ is considered later in Section 3.4.

**3.1. Critical moment**

Fig. 4 shows an example of token flows on a T–L plane. All tokens flow from left to right over time. The scheduling algorithm selects up to M tokens from at most N active tokens and they flow diagonally down. The others which are not selected take horizontal paths. When the event-C or -B happens, denoted by $t_j$ where $0 < j < f$, the LSA is invoked to make a scheduling decision.

**Definition 3 (Local Utilization).** The local utilization $r_i(t_j)$ for a task $T_i$ at time $t_j$ is defined to be $l_i(t_j)/t_f - t_j$, which describes how much processor capacity needs to be utilized for executing $T_i$ within the remaining time until $t_j$. Here, $l_i(t_j)$ is the local remaining execution time of task $T_i$ at time $t_j$. Note that when $k$ is omitted, it implicitly means the current kth T–L plane.

**Theorem 4 (Initial Local Utilization Value in T–L Plane).** Let all tokens arrive at the bottom line before $t_f^k$ on the $(k-1)$th T–L plane. Then, the initial local utilization value $r_i(0) \leq \mu_i$ for all task $T_i$ on the kth T–L plane.

**Proof.** If all tokens arrive at $t_f^k - 1$ with $l_i(t_f^k - 1) \leq 0$, then they can restart on the next T–L plane (the kth T–L plane) from or below the positions where their ideal fluid schedule lines start. The slope of the fluid schedule path of task $T_i$ is $\mu_i$. Thus, $r_i(0) = l_i(0)/t_f \leq \mu_i$.

Well-controlled tokens have both departing and arriving points which are the same as, or lower than, those of their fluid schedule lines on the T–L plane (even though their actual paths on the T–L plane are different from their fluid schedule paths). Well-controlled tokens are locally feasible. Note that we assume $u_i \leq 1$ and $\sum u_i \leq M$.

We define critical moment to be the sufficient and necessary condition that all tokens are not locally feasible. (Local infeasibility of the tokens implies that all tokens do not arrive at the bottom line of the T–L plane before $t_f^k$.) A critical moment is the first secondary event time when more than M tokens simultaneously reach the NLLD. Fig. 5 shows this. Right after the critical moment, only M tokens from those on the NLLD are selected. The non-selected ones move out of the triangle, and as a result, they will not arrive at the bottom line of the T–L plane before $t_f$. Note that only horizontal and diagonal moves are permitted for tokens on the T–L plane.

**Theorem 5 (Critical Moment).** At least one critical moment occurs if and only if tokens are not locally feasible on the T–L plane.

**Proof.** We prove both the necessary and sufficient conditions.

*Only-if part.* Assume that a critical moment occurs. Then, non-selected tokens move off of the T–L plane. Since only $i = 1$ and 0 are allowed for the slope of the token paths, it is impossible that the tokens off of the T–L plane reach the bottom line of the T–L plane.

*If part.* We assume that when tokens are not locally feasible, no critical moment occurs. If there is no critical moment, then the
number of tokens on the NLLD never exceeds \(M\). Thus, all tokens on the diagonal can be selected by the LSA up to time \(t_j\). This contradicts our assumption that tokens are not locally feasible. \(\square\)

We define total local utilization at the \(j\)th secondary event.

**Definition 6** (Total Local Utilization). The total local utilization at the \(j\)th secondary event, \(S(t_j)\), is defined as \(\sum_{i=1}^{N} r_i(t_j)\).

**Corollary 7** (Total Local Utilization at Critical Moment). Suppose that the critical moment occurs at the \(j\)th secondary event on a T–L plane, the total local utilization at the critical moment \(S(t_j)\) is greater than \(M\).

**Proof.** The local remaining execution time \(l_i(t_j)\) for the tasks on the NLLD at the critical moment (at the \(j\)th secondary event) is \(t_j - t_r\), because the T–L plane is an isosceles triangle. Therefore, \(S(t_j) = \sum_{i=1}^{N} r_i(t_j) = \sum_{i=1}^{N} \frac{l_i(t_j)}{t_r} = \sum_{i=1}^{N} \frac{l_i(t_j)}{t_r} = M + \sum_{i=M+1}^{N} \frac{l_i(t_j)}{t_r} > M\). \(\square\)

From the task perspective, the critical moment is the time when more than \(M\) tasks have no local laxity. Thus, the scheduler cannot make them locally feasible with \(M\) processors.

### 3.2. Event-C

Event-C happens when a non-selected token reaches the NLLD. Note that selected tokens never reach the NLLD. Event-C indicates that the task has no local laxity and hence should be selected immediately. Fig. 6 illustrates this, where event-C happens at time \(t_c\) when the token \(T_{M+1}\) reaches the NLLD.

Note that this is under the basic assumption that there are more than \(M\) tasks, i.e., \(N \geq M\). This implicit assumption holds in Sections 3.2 and 3.3.

For convenience, we suppose that an LSA selects \(M\) tasks from \(T_1\) to \(T_M\) and their tokens move diagonally. We give indices \(i\) to tokens inversely according to their local utilization—i.e., \(r_i(t_j) \geq r_{i+1}(t_j)\) where \(1 \leq i < N\) and \(j\), as shown in Fig. 6. It also implies that for all \(M + 1 \leq i < N\) and \(j\), \(r_i(t_j) \geq r_{i+1}(t_j)\).

**Lemma 8** (Sufficient and Necessary Condition for Event-C). The event-C occurs at time \(t_c\), if and only if \(1 - r_{M+1}(t_{c-1}) \leq r_M(t_{c-1})\), where \(r_i(t_{c-1}) \geq r_{i+1}(t_{c-1})\), \(1 \leq i < M\) or \((M + 1 \leq i < N)\).

**Proof.** The time when \(T_{M+1}\) reaches the NLLD is \(t_{c-1} + (l_j - t_{c-1} - l_{M+1}(t_{c-1}))\). The time when the token \(T_M\) reaches the bottom of the T–L plane is \(t_{c-1} + l_M(t_{c-1})\). We prove both the sufficient and necessary conditions. If \(\forall j\), we assume \(1 - r_{M+1}(t_{c-1}) \leq r_M(t_{c-1})\).

1. \(1 - l_{M+1}(t_{c-1}) \leq l_M(t_{c-1})\)
2. \(l_j - t_{c-1} - l_{M+1}(t_{c-1}) \leq l_M(t_{c-1})\)

When adding \(t_{c-1}\) to both sides, we confirm that the time when \(T_{M+1}\) reaches the NLLD is prior or equal to the time when \(T_M\) reaches the bottom of the T–L plane. Thus, event-C then occurs.

**Theorem 10** (Total Local Utilization for Event-C). When event-C occurs at time \(t_c\) and \(S(t_{c-1}) \leq M\), then \(S(t_c) \leq M\), \(\forall c \leq f\).

**Proof.** We define \(t_c - t_{c-1} = (t_j - t_{c-1} - l_{M+1}(t_{c-1}))\) as \(\delta\). Total local remaining execution time at \(t_{c-1}\) is \(\sum_{i=1}^{N} l_i(t_{c-1}) = (t_j - t_{c-1})S(t_{c-1})\) and it decreases by \(M \times \delta\) at \(t_c\) as \(M\) tokens move diagonally. Therefore, \(S(t_{c-1}) \geq M \times r_M(t_{c-1})\).

Based on Lemma 8, \(S(M \times r_M(t_{c-1})) \geq M \times (1 - r_{M+1}(t_{c-1}))\). \(\square\)

3.3. Event-B

Event-B happens when a selected token reaches the bottom side of the T–L plane. Note that non-selected tokens never reach the bottom. Event-B indicates that the task has no local remaining execution time, so it would be better to give the processor time to another task for execution.

Event-B is illustrated in Fig. 8, where it happens at time \( t_b \) when the token of \( T_M \) reaches the bottom. As we do for the analysis of event-C, we suppose that an LSA selects M tasks from \( T_1 \) to \( T_M \) and we give indices \( i \) to tokens inversely according to their local utilization—i.e., \( r_i(t_i) \) is \( r_i+1(t_i) \), where \( 1 \leq i < M \). It also implies that \( r_i(t_i) \geq r_i+1(t_i) \), where \( M + 1 \leq i < N \) and \( \forall j \).

Lemma 11 (Sufficient and Necessary Condition for Event-B). Event-B occurs at time \( t_b \), if and only if \( 1 - r_{m+1}(t_{c-1}) \geq r_i(t_{b-1}), (1 \leq i < M) \) or \( M \leq i < N \).

Proof. The time when \( T_M \) reaches the bottom and the time when \( T_{M+1} \) reaches the NLLD are, respectively, \( t_{b-1} + \hat{b}_M(t_{b-1}) \) and \( t_{b-1} + (t_f - t_{b-1} - \hat{b}_{M+1}(t_{b-1})) \). We prove both sufficient and necessary conditions.

If part. We assume \( r_M(t_{b-1}) \leq 1 - r_{M+1}(t_{b-1}) \).

\[
\begin{align*}
\frac{\hat{b}_M(t_{b-1})}{t_f - t_{b-1}} & \leq 1 - \frac{\hat{b}_{M+1}(t_{b-1})}{t_f - t_{b-1}} \\
\frac{\hat{b}_M(t_{b-1})}{t_f - t_{b-1}} & \leq t_f - t_{b-1} - \hat{b}_{M+1}(t_{b-1}).
\end{align*}
\]

When adding \( t_{b-1} \) to both sides, we confirm that the time when \( T_M \) reaches the bottom is prior to the time when \( T_{M+1} \) reaches the NLLD, which is event-B.

Only-if part. If the secondary event at time \( t_b \) is event-B, then token \( T_M \) must reach the bottom earlier than when token \( T_{M+1} \) reaches the NLLD.

\[
\begin{align*}
t_{b-1} + \hat{b}_M(t_{b-1}) & \leq t_{b-1} + (t_f - t_{b-1} - \hat{b}_{M+1}(t_{b-1})), \\
\frac{\hat{b}_M(t_{b-1})}{t_f - t_{b-1}} & \leq 1 - \frac{\hat{b}_{M+1}(t_{b-1})}{t_f - t_{b-1}}.
\end{align*}
\]

Thus, \( r_M(t_{b-1}) \leq 1 - r_{M+1}(t_{b-1}) \). □

Corollary 12 (Necessary Condition for Event-B). Event-B occurs at time \( t_b \) only if \( S(t_{b-1}) > M \cdot r_M(t_{b-1}) \), where \( r_i(t_{b-1}) \geq r_{i+1}(t_{b-1}) \), \( (1 \leq i < M) \) or \( (M + 1 \leq i < N) \).

Proof.

\[
S(t_{b-1}) = \sum_{i=1}^{M} r_i(t_{b-1}) + \sum_{i=M+1}^{N} r_i(t_{b-1}) > \sum_{i=1}^{M} r_i(t_{b-1}) \geq M \cdot r_M(t_{b-1}).
\]

Thus, \( S(t_{b}) \)

\[
S(t_{b}) = \frac{1}{1 - r_M(t_{b-1})} S(t_{b-1}) - \frac{r_M(t_{b-1})}{1 - r_M(t_{b-1})} M.
\]

Equation (2) is a linear equation as shown in Fig. 9.

According to Corollary 12, when event-B occurs, \( S(t_{b-1}) > M \cdot r_M(t_{b-1}) \), and since we also assume \( S(t_{b-1}) \) is \( M \cdot r_M(t_{b-1}) \), we only consider the range from \( M \cdot (1 - r_{M+1}(t_{c-1})) \) to \( M \) on the x-axis in Fig. 9. Therefore, \( S(t_{b}) \) is \( M \).

3.4. Local feasibility

We now establish LSAs’ local feasibility on the T–L plane based on our previous results.

In Sections 3.2 and 3.3, we suppose that \( N > M \). When less than or equal to \( M \) tokens only exist, they are always locally feasible by an LSA on the T–L plane.

Fig. 9. Linear equation for event-B.

Fig. 10. Token flow when \( N \leq M \).

**Theorem 14** (Local Feasibility with Small Number of Tokens). When \( N \leq M \), tokens are always locally feasible by an LSA.

*Proof.* We assume that if \( N \leq M \), then tokens are not locally feasible by an LSA. If there is not local feasibility, then there should exist at least one critical moment on the T–L plane by Theorem 5. Critical moment implies at least one non-selected token, which contradicts our assumption since all tokens are selectable according to (GLS.1).

Theorem 14 is illustrated in Fig. 10. When the number of tasks is less than the number of processors, an LSA can select all tasks and execute them until their local remaining execution times become zero.

We also observe that at every event-B, the number of active tokens is decreasing. In addition, the number of event-Bs in this case is at most \( N \), since it cannot exceed the number of tokens. Another observation is that event-C never happens when \( N \leq M \), since all tokens are selectable and move diagonally.

Now, we discuss the local feasibility when \( N > M \).

**Theorem 15** (Local Feasibility with Large Number of Tokens). When \( N > M \), tokens are locally feasible by an LSA when \( S(t_0) \leq M \).

*Proof.* We prove this by induction, based on Theorems 10 and 13. Those theorems show that if \( S(t_{j-1}) \leq M \), then \( S(t_j) \leq M \), where \( j \) is the moment when secondary events occur. Since we assume that \( S(t_0) \leq M \), \( S(t_j) \) for all \( j \) never exceeds \( M \) at any secondary event including event-Cs and event-Bs. When \( S(t_j) \) is at most \( M \) for all \( j \), there should be no critical moment on the T–L plane, according to the contrapositive of Corollary 7. By Theorem 5, there being no critical moment implies that they are locally feasible.

When \( N (> M) \) number of tokens are on the T–L plane and their \( S(t_0) \) is at most \( M \), event-C and -B occur without any critical moment according to Theorem 15. Whenever event-B happens, the number of active tokens decreases until there are \( M \) remaining active tokens. Then, according to Theorem 14, all tokens are selectable so that they arrive at the bottom line of the T–L plane with consecutive event-Bs.

Fig. 11 shows one example of T–L plane-based schedules. Six tasks and two processors are assumed for the example, and the local remaining execution time of each task within the time interval \((0, 5)\) is given by the table. Whenever a scheduling decision should be made, we assume that a T–L scheduling algorithm selects tasks to execute according to the GLS. If there is no task that ought to be selected by the GLS, the scheduler selects any task randomly. Note that the GLS is sufficient to provide local feasibility even with this restricted random selection. The T–L plane shows the flow of each token. The figure also shows when event-B and event-C occur over time as denoted by the small filled squares.

At time 0, we assume \( T_3 \) and \( T_2 \) are selected. At time 1, \( T_3 \) generates an event-B by reaching the bottom line and then, we assume \( T_2 \) is selected, based on (GLS.1). At time 3, \( T_1 \) and \( T_2 \) invoke event-B while \( T_4 \) invokes an event-C. According to (GLS.2), \( T_4 \) should be selected immediately. Simultaneously, one more task should be selected by (GLS.1). We assume \( T_2 \) is selected. At time 4, \( T_3 \) and \( T_6 \) generate an event-B and event-C respectively. Thus, \( T_6 \) should be immediately selected according to (GLS.2).

### 4. T–L plane-based scheduling algorithm

Recall that T–L plane-based scheduling consists of two phases: LD-P where each task’s local remaining execution time within a single T–L plane is determined and LS-P where all tasks’ local remaining execution times within a single T–L plane are consumed.

As long as \( S(t_0) \) is less than or equal to \( M \) as in Theorem 15, any LSA designed under the GLS’s is applicable to LS-P. For LD-P, on the other hand, Fig. 2 illustrates that the local remaining execution time at time \( t_0 \) (= 0) for each task, \( l(t_0) \), is determined from the fluid schedule line of each task. Thus, \( l(t_0) \) is either an integer or a real number. However, due to the hardware characteristics of actual processors, \( l(t_0) \) should be an integral multiple of the resolution of the highest precision timer, which makes LD-P more difficult. Besides, in LD-P, we should ensure that all \( S(t_0) \) of continuous T–L planes are at most \( M \) while we determine \( l(t_0) \) of each task to be an integer. Since \( l(t_0) \) of each token is an integer and only diagonal and horizontal moves are permitted, all \( l(t_j) \) (where \( 0 \leq j \leq f \)) are also integers.

We present T–L plane-based scheduling algorithms that use a part of Zhu’s algorithm in [12] for LD-P and use an LSA for LS-P. The design objective here is to make the computational complexity of the algorithms as low as possible. To do so, we suppose that the run-time (e.g., an operating system kernel) offers the scheduler the list of tasks that are in the ready state. In addition, the kernel notifies the scheduler of the task’s identifier (or its pointer) when the event-C or B occurs. More details are provided in this section.

#### 4.1. A non-work-conserving scheduling algorithm

A T–L plane-based scheduling algorithm is shown in Algorithm 1. The key idea of the algorithm is to use two linked lists—e.g., \( \xi \) and \( Q_{\text{run}} \). \( \xi \) is for tasks on the T–L plane and \( Q_{\text{run}} \) is a list of running tasks.

All tasks causing an event-C by reaching the NLLD are inserted (or pushed) into the front of \( Q_{\text{run}} \), as in lines 9–14, and the other
Theorem 15: Computational cost. Note that the number of tasks on the NLLD in $Q_{\text{run}}$ running tasks are inserted (or pushed) into the back of $Q_{\text{run}}$. This ensures that at an event-C, the task at the back is not on NLLD and should be removed first, which simply takes $O(1)$ computational cost. Note that the number of tasks on the NLLD does not exceed $M$ according to Theorem 15 and thus, the size of $Q_{\text{run}}$ never exceeds $M$. The replaced task $t_2$ is inserted into the front of $\zeta$ as in line 13, since $t_2$ is still active.

The task causing an event-B turns into the inactive state, and it is removed from $Q_{\text{run}}$ and inserted into the back of $\zeta$. Then, if the task from the front of $\zeta$ is active (or has non-zero local remaining execution time), it is inserted into the front of $Q_{\text{run}}$. If the task from the front of $\zeta$ is inactive (or has zero local remaining execution time), it is inserted into the back of $\zeta$.

At the period boundary of each task, all tasks’ local remaining execution times, $l_i$, are determined by calling decideLocalParameters($\zeta$), which is the LD-P step. As proved in Theorem 15, LD-P should determine $l_i$ of all tasks to satisfy the inequality, $S(t_0) \leq M$. We adopt a part of Zhu’s algorithm (or BF algorithm) in [12] for LD-P. The BF algorithm allocates some mandatory time units to each task, and one optional time unit to the eligible tasks at each boundary time (i.e., for each period of the task). Here, each task’s $l_i$ is defined as the sum of the mandatory time unit and the optional time unit by the BF algorithm. For an example, suppose that there are six tasks and two processors. Each task is characterized by its execution time and deadline, $T_1 = (2, 5), T_2 = (3, 15), T_3 = (3, 15), T_4 = (2, 6), T_5 = (20, 30)$, and $T_6 = (6, 30)$. Their hyper-period, obtained as least common multiple (LCM), is 30. After time 0, the next task period occurs at 5. Based on Zhu’s algorithm, each task’s local remaining execution time from time $\zeta$.

![Algorithm 1](https://example.com/algorithm1.png)

**Algorithm 1**: A non-work-conserving approach

1. **Input**: $\{(T_1, \ldots, T_n)\}$; tasks in ready state
2. **Output**: Array of dispatched tasks to processors
3. $M$ - # of processors
4. $\zeta$ - Ready queue,
5. $Q_{\text{run}}$ - Queue for running tasks,
6. $Q_{\text{disp}}$ - Maximal size of $Q_{\text{run}}$ is $M$,
7. $l_i$ - Local remaining execution time of $T_i$,
8. $t_1, t_2$ - Temporary variables for tasks
9. **for** event-C **do**
10. $t_1$ = getTaskOfThisEvent();
11. erase($t_1, \zeta$);———-; remove a task $t_1$ from $\zeta$
12. $t_2$ = pop-back($Q_{\text{run}}$);———-; take a task out from the back of $Q_{\text{run}}$
13. push-front($t_2, \zeta$);———-; push a task $t_2$ into the front of $\zeta$
14. push-front($t_1, Q_{\text{run}}$);———-; push a task $t_1$ into the front of $Q_{\text{run}}$
15. **else** if event-B **do**
16. $t_1$ = getTaskOfThisEvent();
17. erase($t_1, Q_{\text{run}}$);———-; remove a task $t_1$ from $Q_{\text{run}}$
18. push-back($t_1, \zeta$);———-; push a task $t_1$ into the back of $\zeta$
19. $t_2$ = pop-front($\zeta$);———-; take a task out from the front of $\zeta$
20. **if** $t_2$ is not NULL **then**
21. **if** $t_2$ has $l_i$ greater than zero **then**
22. push-back($t_2, Q_{\text{run}}$);———-; push a task $t_2$ into the back of $Q_{\text{run}}$
23. **else**
24. push-front($t_2, \zeta$);———-; push a task $t_2$ into the front of $\zeta$
25. **else** if period of each task **do**
26. decideLocalParameters($\zeta$);———-; set all $l_i$’s
27. clear($Q_{\text{run}}$);
28. **for** $i = 1$ to $M$ **do**
29. $t_1$ = pop-front($\zeta$);———-; take a task out from the front of $\zeta$
30. push-back($t_1, Q_{\text{run}}$);———-; push a task $t_1$ into the back of $Q_{\text{run}}$
31. return $Q_{\text{run}}$;

**Fig. 11.** An example schedule by T–L plane-based approaches.

**Fig. 12.** Operations of Algorithm 1.
0 to 5 is computed as 2, 1, 1, 2, 3, and 1. On the first T-L plane, these local remaining execution times should be consumed. The next task period is at time 6. From time 5 to 6, each task’s local remaining execution time is determined to be 1, 0, 0, 0, 1, and 0, all of which should be consumed during time 5–6. This is the same example in [12]. A more detailed description of the BF algorithm is found in [12]. After calling decideLocalParameters(), \( Q_{\text{run}} \) and \( \zeta \) are initialized as shown in line 25 to 28.

During implementation, events C and B are invoked by timer expiration. At the period boundary of each task, algorithm 1 determines \( l_i \), the ith task by calling decideLocalParameters(). As \( l_i \) is determined, the local laxity, \( (t_i - l_i) \) can be easily obtained. Then, the selected tasks start consuming \( l_i \), and simultaneously the non-selected tasks start consuming their local laxities, \( (t_i - l_i) \), where \( t_i \) is the current time. Thus, the subsequent scheduling event occurs at \( \min\{\min(l_i), \min(t_i - l_i)\} \), where i is the index for selected tasks and j is the index for non-selected tasks. Every time the secondary event occurs, the next secondary event time can be determined in the same way after updating each task’s \( l_i \) and \( (t_i - l_i) \). The update computational complexity is \( O(N) \). We assume that \( l_i \) and \( (t_i - l_i) \), two of the states of tasks, are contained in data structures in the kernel, e.g., process control blocks in Real-time Linux, for the kernel to easily maintain.

Since a token remains on the bottom line after event-B though its remaining execution time (not its local remaining execution time) is not zero, the algorithm is a non-work-conserving scheduling one, which implies that the processors may be idle even when tasks are present in the ready queue and some processors are available. We consider a work-conserving scheduling algorithm in the following section.

The time complexity of this algorithm depends on the operations on \( Q_{\text{run}} \) and \( \zeta \) as well as on the function decideLocalParameters(). If these are implemented as linked lists, the time complexities of all insert (push) and delete (pop) operations are \( O(1) \). The operation erase() also costs \( O(1) \), since we assume that the kernel notifies the scheduler of the tasks’ identifiers (or its pointer) when the event-C or -B occurs. Thus, the time complexities of the algorithm are \( O(1) \) for event-C or event-B. As for the time complexity of decideLocalParameters() and the for-loop in line 26, the time complexity of the BF algorithm is known to be \( O(N) \) in [12] and, therefore, \( O(\max(N, M)) \) is the time complexity of the entire algorithm.

4.2. A work-conserving scheduling algorithm

Given the fact that obeying the GLS’s is sufficient for a scheduling algorithm to provide local feasibility in a T-L plane, Algorithm 1 can be modified to be work-conserving as shown in Algorithm 2. Note that it is an example of the locally feasible scheduling algorithms. As long as it obeys GLS’s, any priority assignment can be used.

Algorithm 2 uses three linked lists—e.g., \( \xi_0, \xi_k+1, \) and \( Q_{\text{run}} \). \( \xi_0 \) is for tasks on the current T-L plane and \( \xi_{k+1} \) is for tasks on the next T-L plane. \( \xi_0 \) and \( \xi_{k+1} \) are switched at the period boundaries of all tasks. \( Q_{\text{run}} \) is a list for running tasks.

In Algorithm 2, we add \( e_i \) to represent the actual remaining execution time of \( T_i \). \( e_i \) is compared with \( C_i \) that is updated after LD-P in line 26. \( C_i \) represents the remaining execution time after \( T_i \) consumes its allocated time by LD-P. The work-conserving policy that allows no processor iddle time implies that \( e_i \) and \( C_i \) could be different. If \( e_i \) is greater than \( C_i \), the token of \( T_i \) is active so the task should be inserted into the front of \( \xi_{k+1} \) as in line 32. Otherwise, it is inserted into the back of \( \xi_{k+1} \). Therefore, after every task’s period, all active tasks are collected at the back. Even after all tokens in \( \xi_{k} \) become inactive, a task can be moved into \( Q_{\text{run}} \) to run if some processors are available, as shown in line 23, which makes it work-conserving. The time complexity of Algorithm 2 is \( O(\max(N, M)) \), the same as that of Algorithm 1. Table 1 is the summary of time complexities of the scheduling algorithms described in this paper.

Algorithm 2: A Work-Conserving Algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textbf{Input:} ( T = {T_1, ..., T_n} ), tasks in ready state</td>
</tr>
<tr>
<td>2</td>
<td>\textbf{Output:} Array of dispatched tasks to processors</td>
</tr>
<tr>
<td>3</td>
<td>\textbf{M-#} of processors</td>
</tr>
<tr>
<td>4</td>
<td>( \xi_0, \xi_{k+1} ) - Ready queues,</td>
</tr>
<tr>
<td>5</td>
<td>( Q_{\text{run}} ) - Queue for running tasks,</td>
</tr>
<tr>
<td>6</td>
<td>( M ) - Maximal size of ( Q_{\text{run}} ) is ( M ),</td>
</tr>
<tr>
<td>7</td>
<td>( l_i ) - Local remaining execution time of ( T_i ),</td>
</tr>
<tr>
<td>8</td>
<td>( c_i ) - Remaining execution time of ( T_i ),</td>
</tr>
<tr>
<td>9</td>
<td>( e_i ) - Actual remaining execution time of ( T_i ),</td>
</tr>
<tr>
<td>10</td>
<td>( t_1, t_2 ) - Temporary variables for tasks</td>
</tr>
<tr>
<td>11</td>
<td>if event-C then</td>
</tr>
<tr>
<td>12</td>
<td>( t_1 = \text{getTaskOfThisEvent}() );</td>
</tr>
<tr>
<td>13</td>
<td>erase(( t_1, \xi_{k+1} )); ————; remove a task ( T_i ) from ( \xi_{k+1} )</td>
</tr>
<tr>
<td>14</td>
<td>( t_2 ) = \text{pop-back}(Q_{\text{run}}); ————; take a task out from the back of ( Q_{\text{run}} )</td>
</tr>
<tr>
<td>15</td>
<td>\text{push-front}(( t_2, \xi_0 )); ————; push a task ( T_i ) into the front of ( \xi_0 )</td>
</tr>
<tr>
<td>16</td>
<td>\text{push-front}(( t_1, Q_{\text{run}} )); ————; push a task ( T_i ) into the front of ( Q_{\text{run}} )</td>
</tr>
<tr>
<td>17</td>
<td>else if event-B then</td>
</tr>
<tr>
<td>18</td>
<td>( t_1 = \text{getTaskOfThisEvent}() );</td>
</tr>
<tr>
<td>19</td>
<td>erase(( t_1, Q_{\text{run}} )); ————; remove a task ( T_i ) from ( Q_{\text{run}} )</td>
</tr>
<tr>
<td>20</td>
<td>\text{push-back}(( t_1, \xi_{k+1} )); ————; push a task ( T_i ) into the back of ( \xi_{k+1} )</td>
</tr>
<tr>
<td>21</td>
<td>( t_2 ) = \text{pop-front}(( \xi_0 )); ————; take a task out from the front of ( \xi_0 )</td>
</tr>
<tr>
<td>22</td>
<td>if ( t_2 ) is not \text{NULL} then</td>
</tr>
<tr>
<td>23</td>
<td>\text{push-back}(( t_2, Q_{\text{run}} )); ————; push a task ( T_i ) into the back of ( Q_{\text{run}} )</td>
</tr>
<tr>
<td>24</td>
<td>else if periods of each task then</td>
</tr>
<tr>
<td>25</td>
<td>attach(Q_{\text{run}}, ( \xi_{k+1} )); ————; attach ( Q_{\text{run}} ) to ( \xi_{k+1} )</td>
</tr>
<tr>
<td>26</td>
<td>decideLocalParameters(( \xi_{k+1} )); ———; set all ( C_i )'s</td>
</tr>
<tr>
<td>27</td>
<td>for ( i = 1 ) to ( N ) do update(( C_i ));</td>
</tr>
<tr>
<td>28</td>
<td>for ( i = 1 ) to ( N ) do update(( \xi_{k+1} ));</td>
</tr>
<tr>
<td>29</td>
<td>for ( i = 1 ) to ( N ) do</td>
</tr>
<tr>
<td>30</td>
<td>( t_1 ) = \text{pop-front}(( \xi_{k+1} )); ————; take a task out from the front of ( \xi_{k+1} )</td>
</tr>
<tr>
<td>31</td>
<td>if ( e_i ) of ( T_i ) is greater than its ( C_i ) then</td>
</tr>
<tr>
<td>32</td>
<td>\text{push-front}(( t_1, \xi_{k+1} )); ————; push a task ( T_i ) into the front of ( \xi_{k+1} )</td>
</tr>
<tr>
<td>33</td>
<td>else ( \text{push-back}(( t_1, \xi_{k+1} )); ————; push a task ( T_i ) into the back of ( \xi_{k+1} )</td>
</tr>
<tr>
<td>34</td>
<td>switchReadyQueue(( \xi_{k+1} ), ( \xi_{k+1} ));</td>
</tr>
<tr>
<td>35</td>
<td>clear(Q_{\text{run}});</td>
</tr>
<tr>
<td>36</td>
<td>for ( i = 1 ) to ( N ) do</td>
</tr>
<tr>
<td>37</td>
<td>( T_i ) = \text{pop-front}(( \xi_{k+1} )); ————; take a task out from the front of ( \xi_{k+1} )</td>
</tr>
<tr>
<td>38</td>
<td>\text{push-back}(( T_i, Q_{\text{run}} )); ————; push a task ( T_i ) into the back of ( Q_{\text{run}} )</td>
</tr>
<tr>
<td>39</td>
<td>return ( Q_{\text{run}} );</td>
</tr>
</tbody>
</table>

4.3. Algorithm overhead

One of the main concerns against global scheduling algorithms for multiprocessors is their overhead caused by frequent scheduler invocations. In [11], Srinivasan et al. identify three specific overheads:

Table 1: Time complexity.

<table>
<thead>
<tr>
<th>Scheduling algorithms</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhu’s algorithm [12]</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>Non-work-conserving Algorithm</td>
<td>( O(\max(N, M)) )</td>
</tr>
<tr>
<td>Work-conserving Algorithm</td>
<td>( O(\max(N, M)) )</td>
</tr>
</tbody>
</table>
(1) **Scheduling overhead**, which accounts for the time spent by the scheduling algorithm, including that for constructing schedules and ready-queue operations;

(2) **Context-switching overhead**, which accounts for the time spent in storing the preempted task's context and loading the selected task's context; and

(3) **Cache-related preemption delay**, which accounts for the time incurred in recovering from cache misses that a task may suffer when it resumes after a preemption.

Note that when a scheduler is invoked, the context-switching overhead and cache-related preemption delay may not happen. Srinivasan et al. also show that the number of task preemptions can be bounded by observing that when a task is scheduled (selected) consecutively for execution, it can be allowed to continue its execution on the same processor. This reduces the number of context-switches and possibility of cache misses. They bound the number of task preemptions under *P*fair, illustrating how much a task's execution time inflates due to the aforementioned overhead. They show that for *P*fair, the overhead depends on the time quantum size.

In contrast to *P*fair, T–L plane-based scheduling is free from time quanta. Here, we use the number of scheduler invocations
as a metric for overhead measurement, since it is the scheduler invocation that contributes to all three of the overheads previously discussed. We now derive an upper bound for the scheduler invocations under T-L plane-based scheduling.

**Theorem 16 (Upper Bound on Number of Secondary Events in T-L Plane).** When tokens are locally feasible on the T-L plane by Algorithm 1, the number of events on the plane is bounded within \( \min(N + 1, \tau_t / \tau_{sys}) \), where \( \tau_{sys} \) is the system clock tick.

**Proof.** We consider two possible cases, when a token \( T_t \) reaches the NLLD, and when it reaches the bottom. After \( T_t \) reaches the NLLD, it will move along the diagonal to the rightmost vertex of the T-L plane, because we assume that the tasks are locally feasible. In this case, \( T_t \) raises one secondary event, event-C. Note that its final event-B at the rightmost vertex occurs together with the next event of another task’s period (i.e., beginning of the new T-L plane).

If \( T_t \) reaches the bottom, then the token becomes inactive and will arrive at the right vertex after a while. In this case, it raises one secondary event, event-B. Therefore, each \( T_t \) can cause one secondary event on the T-L plane. Thus, \( N \) number of tokens can cause \( N + 1 \) number of events, which includes \( N \) secondary events and a task’s period boundary at the rightmost vertex, when the system clock tick allows it. When \( N + 1 \) is greater than \( \tau_t / \tau_{sys} \), only \( \tau_t / \tau_{sys} \) number of secondary events can occur since the \( i \)'s of all tokens are determined as integers by LD-P. □

The overhead of Algorithm 2 has a corresponding property except that early task completions should be considered in a work-conserving scheduling algorithm. However, it is still true that the number of secondary events linearly depends on \( N \) because the number of task completions in a T-L plane cannot exceed the number of \( N \).

**Theorem 17 (Upper Bound of the Number of T-L Planes Over Time).** When tasks can be feasibly scheduled by Algorithm 1, an upper bound of the number of T-L planes in a time interval \([t_i, t_f]\) is:

\[
1 + \sum_{i=1}^{N} \left\lfloor \frac{t_f - t_i}{p_i} \right\rfloor,
\]

where \( p_i \) is the period of \( T_i \).

**Proof.** Each T-L plane is constructed between two consecutive occurrences of task period boundaries, as shown in Section 2.2. The number of task period boundaries during the time between \( t_i \) and \( t_f \) is \( \sum_{i=1}^{N} \left\lfloor \frac{t_f - t_i}{p_i} \right\rfloor \). The number of T-L planes in a time interval is at most one more than the number of task period boundary occurrences. Thus, there can be at most 1 + \( \sum_{i=1}^{N} \left\lfloor \frac{t_f - t_i}{p_i} \right\rfloor \) number of T-L planes between \( t_i \) and \( t_f \). □

Theorems 16 and 17 show that the number of scheduler invocations of Algorithm 1 is primarily dependent on \( N \) and \( p_i \)—i.e., more tasks or shorter periods of tasks result in increased overhead. In other words, T-L plane-based approaches have overhead similar to that of many other traditional real-time scheduling algorithms. The overhead of EDF, for example, also increases as the number of tasks grows or the periods of tasks become shorter.

The number of scheduler invocations of the PFair algorithm depends upon the time quantum size \( Q \). In a time interval \([t_i, t_f]\), the number of PFair scheduler invocations is \( \frac{t_f - t_i}{Q} \). Thus, the overhead of PFair does not depend on \( N \) or each \( p_i \). When the time quantum becomes shorter, the overhead of PFair increases as opposed to T-L plane-based approaches.

### 4.4 Comparison

T-L plane-based scheduling consists of two phases, LD-P and LS-P. We propose LSAs for LS-P and adopt a part of Zhu’s algorithm for LD-P. Zhu’s algorithm utilizes McNaughton’s rule for LS-P. Therefore, we first compare LSAs with McNaughton’s rule for LS-P. Subsequently, we describe several properties of Zhu’s algorithm for LD-P, which our algorithms partly inherit for LD-P.

McNaughton’s rule that Zhu et al. uses for local scheduling was originally proposed to minimize the schedule length [10].

This scheduling problem is denoted as \( R_{\text{Pmtn}}(\text{max}) \). The minimal schedule length \( D \) is given as \( \max\{\sum_{i=0}^{N} f_i(T_i) / M, \max(f_i(T_i))\} \) within a local schedule by McNaughton’s rule, where \( N \) is the number of tokens and \( M \) is the number of processors. After ordering tasks arbitrarily, tasks are assigned to a processor to fill up to time \( D \) before tasks start being assigned to another processor.

Whereas McNaughton’s rule provides the minimal schedule length, it may cause unnecessary migration for some cases. For example, when three processors and three tasks are given and each task’s \( f_i(T_i) = 3, 2 \) within a time interval \([0, 3]\), \( D \) is calculated as 3. McNaughton’s rule assigns \( T_1 \) to processor 1. Then, it assigns \( T_2 \) to processor 2 and assigns \( T_3 \) to processor 2 to fill up processor 2 to time 3. After filling up processor 2, it assigns the remaining part of \( T_3 \) to processor 3, which consequently causes a migration of \( T_3 \) from processor 2 to processor 3.

In contrast, T-L plane-based LSAs proposed in this paper do not cause this migration for this example, since they would simply assign \( T_3 \) to processor 3 at the beginning. It is because T-L plane-based local scheduling is designed to make all tasks locally feasible. This scheduling problem is denoted as \( R_{\text{Pmtn}|\text{feasible}} \).

Above all, the GLS for LSA’s gives significant flexibility to local scheduling design. For instance, when resource sharing constraints would be considered, the GLS would be a minimum requirement for a possible LSA to satisfy. Again, LSAs for a periodic task set can be designed to follow the GLS, the sufficient condition for local feasibility.

[12] characterizes the BF algorithm against the PFair algorithm with several experimental results. Experimentally, BF has fewer scheduling events than PFair does, especially when the maximum period of tasks becomes longer or the number of tasks becomes smaller. In addition, BF uses less processing time to generate the whole schedule than Pfair when the number of tasks is less than 100 because of fewer events. Our algorithms use BF for LD-P and the comparison against Pfair does not deviate much from [12]. Thus, in the following section, we focus on evaluating our algorithms against BF’s local schedule experimentally, rather than comparing with Pfair.

### 5. Experimental evaluation

We conducted simulation-based experimental studies to compare T-L plane-based scheduling algorithms with some existing algorithms and to validate our analytical results on overhead.

**Comparison with McNaughton’s rule** [10]. Zhu’s algorithm in [12] uses McNaughton’s rule for local scheduling. We compare Algorithm 1 to McNaughton’s rule for local scheduling. We first consider four processors and local scheduling within a time interval \([0, 10]\). We randomly generate tasks to be subject to \( D \) that is given as \([2, 4, 6, 8, 10]\). Several experiments are repeated and we show the results with a 95% confidence interval.

**Fig. 13(a)** shows the number of migrations over varying \( D \). Over all \( D \), our algorithm generates fewer migrations than McNaughton’s rule. However, we observe that McNaughton’s rule always produces the minimum schedule lengths \( D \), while our algorithm’s schedule lengths are a little longer (but never exceed the time interval, 10) in **Fig. 13(b)**. This implies that McNaughton’s rule may incur unnecessary migrations in order to minimize the schedule length, which is its original scheduling objective, as discussed in Section 4.4. We observe the similar results for the case of eight processors in **Fig. 14(a)** and (b). However, the objective of
our scheduling approach is different from McNaughton’s—it is to complete all the tasks by their deadlines.

**Overhead.** To validate our analytical results on overhead, we considered an SMP machine with two processors, and several tasks running on the system. To evaluate the overhead in terms of the number of scheduler invocations, we define the scheduler invocation frequency as the number of scheduler invocations during a time interval: $[t_i, t_j]$ divided by $[t_i, t_j]$. We set $[t_i, t_j]$ as 30 and we also set the system clock tick $\tau_{sys}$ as 1. The total utilization is at most 2, the capacity of processors.

First, we randomly select task periods between 5 and 30 and the number of tasks around 5. Fig. 15(a) shows that the actual frequency increases when the minimum task period $p_{min} = \min\{p_1, \ldots, p_n\}$ becomes shorter. Second, we increase the number of tasks and Fig. 15(b) shows that more running tasks leads to more scheduler invocations. In Fig. 15, the error bar around each data point represents 95% confidence interval of that data point.

As Theorems 16 and 17 indicate, we observe that the smaller $p_{min}$ and $N$ proportionally affect the overhead. Thus, our experimental results validate our analytical results.

6. Conclusions

We introduce a novel abstraction for reasoning about timeliness of independent periodic task execution behavior on multiprocessors, called T–L plane. The abstraction allows viewing multiprocessor scheduling as scheduling on repeatedly occurring T–L planes, and correct scheduling on a single T–L plane leads to an optimal solution for all times. For a single T–L plane, we analytically show a sufficient condition, called GLS, to provide a locally feasible schedule. Based on these, we propose two examples of T–L plane-based real-time scheduling algorithms, which include non-work-conserving and work-conserving approaches. We also establish that the algorithm overhead is bounded in terms of the number of scheduler invocations, which is validated by our experimental (simulation) results. We experimentally show that our T–L plane-based local scheduling approach outperforms an existing approach in terms of the number of task migrations.

We believe that T–L plane-based algorithms are flexible to accommodate more relaxed task models having different aspects of our task model such as arrival pattern, time constraint, execution time, and dependency properties.

Since the scheduling in this paper is under an assumption that the next period time is known beforehand at the current period time, which allows a T–L plane to be established between those two adjacent events, it is not directly applicable to a sporadic task model, for example. For the sporadic task model, the T–L plane can be established between the current period time and the earliest possible time of the next period, and then the T–L plane scheduling algorithm can be applied within the time interval. After the earliest possible time of the next period, a different scheduling algorithm may be needed up to the time when the task actually arrives. Ascertain the validity of that speculation, and making such a scheduling algorithm simple and efficient enough is an important future research direction.

Acknowledgments

This work was supported by the US Office of Naval Research under Grant N00014-00-0-0549 and by The MITRE Corporation under Grant 52917. E. Douglas Jensen’s participation was supported by the MITRE Technology Program. A preliminary version of this paper appeared as “An Optimal Real-Time Scheduling Algorithm for Multiprocessors,” H. Cho, B. Ravindran, and E.D. Jensen, IEEE Real-Time Systems Symposium (RTSS), December 2006.

References


Hyeonjoong Cho is an Assistant Professor in the Department of Computer and Information Science at Korea University. Before he joined Korea University in 2009, he was a senior researcher in Electronics and Telecommunications Research Institute. He received his Ph.D. degree in Computer Engineering from Virginia Polytechnic Institute and State University (Virginia Tech) in 2006. He had worked for Samsung Electronics as a senior software engineer at Factory Automation research institute before pursuing Ph.D. degree. His research focuses on real-time scheduling and synchronization for single and multiprocessors. His other research interests include RTOS, embedded systems, industrial field bus, and algorithm design.

Binoy Ravindran is an Associate Professor in the Department of Electrical and Computer Engineering at Virginia Tech. He joined Virginia Tech in 1998 after receiving his Ph.D. degree in Computer Science from the University of Texas at Arlington (UT). He also received a “Doctoral Dissertation Award” from UT for outstanding doctoral research. The central theme of Dr. Ravindran’s research is adaptive real-time software—i.e., real-time software that can dynamically adapt to uncertainties in its operating environments, and satisfy time constraints with acceptable predictability. His group has recently developed several new results on adaptive real-time computing, including those on multiprocessor/distributed scheduling and synchronization, (mobile ad hoc) network scheduling and routing, energy-efficient scheduling, and memory management and garbage collection. Many of these results have been transitioned for use in the US Department of Defense programs through experimental real-time software systems (including real-time middleware) that his group has developed.

E. Douglas Jensen is the Chief Scientist of the Information Technologies Directorate at MITRE Corporation. His principal focus is on time-critical resource management in dynamic distributed object systems, particularly for combat platform and battle management applications. He directs and conducts research, performs technology transition, and consults on DoD programs. He joined MITRE from previous positions at HP, Digital Equipment, and other computer companies. From 1979 to 1987 he was on the faculty of the Computer Science Department at Carnegie Mellon University, where he created and directed the largest academic real-time research group of its time. Prior to that, he was employed in the real-time computer industry, where he was engaged in research and advanced technology development of distributed real-time computer systems, hardware, and software. He is considered a member of the original pioneerers, and leading visionaries, of distributed real-time computer systems, and is widely sought out throughout the world as a speaker and consultant.